Evaluation Symbolique à Contraintes pour la Validation - Application à Java/JML

Frédéric Dadeau

Université de Franche-Comté **UFR Sciences et Techniques**

Numéro d'ordre : 1155 **École doctorale SPIM**

Évaluation symbolique à contraintes pour la validation

– Application à Java/JML –

THÈSE

présentée et soutenue publiquement le 19 juillet 2006

pour l'obtention du grade de

Docteur de l'université de Franche-Comté

(Spécialité Informatique)

par

Frédéric Dadeau

Composition du jury

Président :	Jacques Julliand, Professeur à l'Université de Franche-Comté
Directeurs :	Bruno Legeard, Professeur à l'Université de Franche-Comté Fabrice Bouquet, Maître de Conférences HDR à l'Université de Franche-Comté
Rapporteurs :	Yves Ledru, Professeur à l'Université Joseph Fourier Grenoble I Christine Paulin-Mohring, Professeur à l'Université de Paris XI
Examinateur :	Bernard Botella, Ingénieur à THALES Systèmes Aéroportés

Laboratoire d'Informatique de l'Université de Franche-Comté — CNRS FRE 2661

Mis en page avec la classe thloria.

Remerciements

Je tiens en premier lieu à remercier mes deux responsables, Bruno Legeard et Fabrice Bouquet, pour la direction qu'ils ont su faire prendre à cette thèse, et les précieux conseils distillés durant ces années. Une grand merci également aux rapporteurs, Christine Paulin-Mohring et Yves Ledru, qui ont montré l'intérêt porté à mes travaux en acceptant cette lourde tâche. Je n'oublie pas non plus Bernard Botella pour ses compétences et son expérience du monde industriel qui ont fait de lui un examinateur de marque. Pour finir, je remercie Jacques Julliand d'avoir accepté de se joindre au jury et d'en avoir pris la place de Président.

Je souhaite également remercier ici les membres du LIFC qui m'ont accueilli à bras ouverts, me faisant passer du statut d'étudiant à celui de collègue, en particulier : Jacques, Françoise, Olga (O.K. vainqueur par K.O. !), Alain et Isabelle. Je salue également mes acolytes du bureau 414 : Pierre-Cyrille et Mathilde, plus tard remplacée par Julien, qui ont contribué à une ambiance détendue (voire très détendue le vendredi) mais sérieuse, qui a grandement favorisé le bon déroulement de mes travaux.

Je profite de l'occasion pour saluer et remercier les personnes qu'il m'a été donné de rencontrer au cours des projets auxquels j'ai participé. Historiquement, je commence par les membres du projet CASSIS à Nancy : Michaël Rusinowitch, Silvio Ranise, David Déharbe, Laurent Vigneron, et Christophe Ringeissen ; puis, les membres de l'ACI GECCOO[1] *: Christine Paulin, Claude Marché, Jean-Christophe Filliâtre, Sylvain Conchon du LRI Orsay, Marie-Laure Potet, Yves Ledru, Sylvain Boulmé, et Nicolas Stouls (inoubliable GO de l'EJCP'04) du LSR-IMAG de Grenoble, Marieke Huisman et Julien Charles de l'INRIA de Sophia-Antipolis ; et, enfin, les membres du projet RNTL DANOCOPS : Bernard Botella, Sylvère Lhermite et Eric Maes de Thalès Systèmes Aéroportés, Jean-François Tillman, Pierre Guestault et Mathieu Watel d'AxLog, Michel Rueher et Hélène Collavizza de l'ESSI de Nice, Ioannis Parissis et Besnik Seljimi du LSR-IMAG. Tous ont contribué, lors des réunions techniques ou pleinières des projets, au cours de diverses présentations et discussions à faire évoluer ces travaux dans la meilleure direction.*

Je n'oublie pas non plus mes compagnons d'infortune, thèsards au LIFC, et plus particulièrement : Yohan, Emilie (courage !), Franck, Julien, Stéphane, Jean-François, Bruno, Fabien, Arnaud, Muriel, et Vincent, ainsi que tous les anciens du laboratoire, ayant à présent intégré l'entreprise Leirios Technologies^{TM} *: Fabien, Nicolas, Stéphane, et Séverine. Tous ont contribué à instaurer une ambiance à la fois détendue et joviale, ponctuée, ça et là, de parties de Uno puis de tarot, et de réflexions profondes sur la condition (in)humaine du doctorant autour d'interminables tournées de cafés. A tous, je vous souhaite de réussir dans tout ce que vous aurez l'audace d'entreprendre, et j'espère que nos chemins auront à nouveau l'occasion de se croiser.*

[1] ayant financé ces travaux

J'ai eu la chance durant cette thèse de pouvoir exercer des activités d'enseignement. Je tiens donc à remercier les membres de l'équipe pédagogique du département informatique qui m'ont confié des responsabilités d'enseignement et tout particulièrement François, pour la liberté et la confiance qu'il m'a témoigné durant ces trois années, ainsi que Jean-Christophe, Olga, Jean-Michel, et enfin Fabrice (qui me doit toujours 11 HTD). J'ai ainsi été amené à côtoyer des étudiants qui laisseront, à moi ou à mes collègues, un souvenir impérissable (cf. N.S.). Je pense notamment à Armel, Benjamin, Romain, Florent, Julien, Guillaume et au reste des deux promos de Master 1 et 2 de l'année scolaire 2005-2006.

Pour finir, je remercie ma famille pour m'avoir supporté dans mes choix de carrière, même les plus incompréhensibles. Enfin, last but not least*, je tiens à rendre hommage à celle qui partage ma vie depuis plus de 7 ans, et qui m'a supporté (dans les deux sens du terme) durant ces trois années de thèse, avant d'accepter de me faire passer du statut de docteur au statut de mari à exactement un mois d'intervalle. Audrey, ma chère épouse, je t'aime.*

À ma famille, qui s'agrandit ...

Table des matières

Table des figures

Liste des tableaux

Première partie

Contexte et problématique

Chapitre 1

Introduction

Sommaire

Les systèmes informatisés sont de plus en plus présents autour de nous. Ils se sont imposés dans notre quotidien à tel point qu'ils se sont rendus indispensables. Néanmoins, tous les systèmes ne sont pas en charge des mêmes responsabilités. Ceux-ci sont classés en fonction des dommages et des coûts qu'ils occasionnent en cas de dysfonctionnement. Ainsi, les systèmes dits *critiques* sont ceux qui mettent en jeu des vies humaines (avions, centrales nucléaires, ...) ou d'importantes sommes d'argent (transactions bancaires, navettes spatiales, ...). Régulièrement, il arrive qu'un système de ce type subisse une panne, due à une défaillance logicielle. On citera parmi les plus récentes la panne du réseau téléphonique de Bouygues Telecom, le 17 novembre 2005, dont le coût est évalué à 16 millions d'euros, la gigantesque panne qui a touché les guichets de vente de billets le 15 juillet 2005, paralysant 800 des 4000 terminaux de vente de la SNCF pendant plus de 24 heures, ou encore l'échec du lancement d'Ariane 5, le 4 juin 1996, qui se solda par l'autodestruction de la fusée, et les conséquences que l'on sait (3 milliards de francs –à l'époque– de pertes, sans compter la mauvaise publicité apportée à l'aérospatiale européenne).

Plus les dommages potentiels sont élevés, plus on exige des systèmes et des logiciels embarqués qu'ils soient sûrs, de manière à assurer l'intégrité des entités qui en dépendent. La conception de systèmes sûrs s'articule autour de deux axes complémentaires : l'utilisation de composants matériels fiables, et le développement de logiciels de qualité. C'est dans ce deuxième cadre que se situe la recherche dans le domaine du *génie logiciel*, qu'elle soit effectuée en milieu universitaire ou en milieu industriel. La réalisation des logiciels critiques est devenue une tâche complexe qui ne laisse pas la place à l'erreur ou à l'incertitude. Pour assister les ingénieurs à la mise au point de ces logiciels, le génie logiciel préconise le recours aux *méthodes formelles* permettant de *vérifier* et de *valider* les systèmes à l'aide de modèles formels.

Les méthodes formelles se fondent sur l'emploi de notations logiques et/ou mathématiques, permettant de décrire le système considéré par un modèle formel, levant ainsi les ambiguïtés induites par le langage naturel décrivant les spécifications du système. La *vérification* consiste à s'assurer de la cohérence d'un modèle tandis que la *validation* s'intéresse à garantir la conformité, soit d'un modèle par rapport aux spécifications des besoins, soit d'un programme par rapport à un modèle formel décrivant son fonctionnement.

Dans tout ce document, nous désignerons sous le terme "les spécifications" les exigences du modèle fonctionnel données par le cahier des charges, et nous désignerons sous le terme "le modèle" ou "la modélisation" une modélisation formelle du système étudié.

Vérification et validation vont de pair et sont les bases du génie logiciel. Ces deux démarches s'appliquent sur les modèles, mais peuvent aussi être mises en œuvre sur des programmes pour s'assurer qu'ils sont conformes à un modèle de référence.

1.1 Validation et vérification formelle des logiciels

Une partie du génie logiciel s'appuie sur la conception de modèles décrits par des notations formelles. Diverses techniques existent pour vérifier un modèle, c'est-à-dire s'assurer de sa cohérence. Néanmoins, si la vérification est une étape importante, la validation du modèle est également primordiale, car elle permet de s'assurer que le modèle, bien qu'il soit cohérent, respecte bien les volontés décrites dans le cahier des charges. Un modèle validé peut alors servir de référence lors de la confrontation entre un programme et sa modélisation, par exemple en utilisant des techniques de test fonctionnel.

1.1.1 Vérification de modèles formels

La vérification d'un modèle consiste à s'assurer que celui-ci respecte des propriétés de cohérence. Ainsi, la vérification s'intéresse à établir que les invariants du système, c'est-à-dire des propriétés que l'on souhaite être toujours vraies, sont vérifiés dans chacun des états atteignables du système. La vérification de ces propriétés est réalisée par la mise en œuvre de techniques telles que le *model-checking* ou la *preuve de théorèmes*.

Le model-checking [CGP99, McM92] est une technique qui consiste à énumérer exhaustivement tous les états de la modélisation par activation des transitions entre ces états, à partir de l'état initial. Au fur et à mesure que les états sont calculés, les propriétés statiques ou dynamiques sont vérifiées. Si un état ne respecte pas une de ces propriétés, la trace d'exécution menant à cet état présente un contre-exemple. Lorsque tous les états ont été calculés, si aucune propriété n'est violée alors le modèle est vérifié. Si cette approche est exhaustive, elle souffre dans la pratique de l'explosion combinatoire du nombre d'états et elle requiert que les données utilisées dans la modélisation soient toutes connues, interdisant ainsi l'introduction de paramètres dans le modèle.

Dans cette optique, les techniques de preuve de théorème [Hoa69] proposent de considérer des formules qui doivent être prouvées comme valides pour garantir la correction du modèle formel. Ces formules, appelées *obligations de preuve*, sont calculées à partir des propriétés (statiques ou temporelles) exprimées dans le modèle. Pour décider de la validité des obligations de preuve, cette approche se base sur l'application de règles de déduction sur les formules d'obligations de preuve, jusqu'à l'obtention d'une formule permettant de conclure. Si elle permet de décider la validité d'une formule, alors l'application de ces règles constitue une procédure de décision. Les techniques de preuve permettent de vérifier des systèmes paramétrés levant ainsi la restriction du model-checking. Néanmoins, elles sont confrontées au problème de la terminaison des algorithmes de déduction, qui sont amenés dans certains cas à diverger. Dans ce cas, aucune conclusion ne peut être tirée sur la validité de la formule car aucune décision n'a pu être effectuée. Ces procédures de décisions sont implantées dans des prouveurs automatiques tels que Simplify [DNS03], ICS [FORS01b], haRV^{ey} [DR03].

Pour pallier ces problèmes, des assistants de preuve permettent de décharger une partie des obligations de preuve automatiquement mais ils requièrent l'intervention et l'expertise d'un utilisateur pour la preuve de certaines sous-formules intermédiaires. On citera parmi les assistants de preuve les plus utilisés : Coq [CDT01], PVS [ORS92] ou Isabelle [Pau94].

Cependant, un modèle vérifié n'est pas pour autant valide car la vérification ne garantit d'aucune manière que le système modélisé se comporte selon les exigences du cahier des charges. L'étape de validation est donc nécessaire à la mise au point d'un logiciel sûr.

1.1.2 Validation de modèles formels

Valider un modèle consiste à s'assurer du bon comportement de celui-ci. Le spécifieur cherche ainsi à s'assurer que la modélisation qu'il a écrite est conforme au cahier des charges. Une technique de validation courante consiste à animer le modèle, c'est-à-dire à simuler son exécution [Kne89]. Il s'agit d'un processus semi-automatique qui requiert l'intervention humaine pour choisir les opérations du système à exécuter, saisir les valeurs des éventuels paramètres de l'opération, et enfin –et surtout– comparer les résultats produits par le modèle avec les descriptions contenues dans le cahier des charges.

La validation d'un modèle peut également être réalisée par la vérification de propriétés temporelles spécifiques, comme les propriétés de sûreté (garantissant que quelque chose de mauvais ne doit jamais arriver), de vivacité (quelque chose de bon doit fatalement arriver),

d'atteignabilité (une certaine situation doit être atteinte), d'absence de blocage (le système peut toujours progresser), et d'équité (sous certaines conditions, quelque chose aura lieu un nombre infini de fois). L'expression de ces propriétés temporelles est laissée au soin de l'utilisateur, qui décrit les propriétés qu'il souhaite voir satisfaites sur l'ensemble du système modélisé. Ce processus s'apparente également à la validation d'un modèle formel puisqu'il vise à s'assurer du bon comportement du système. La vérification de propriétés temporelles implique la mise en œuvre de techniques de "model-checking" basées sur la vérification de la propriété considérée sur le graphe d'atteignabilité du système. Néanmoins, l'animation et la vérification de propriétés temporelles sont vues comme complémentaires : l'animation donne au spécifieur l'aperçu de l'exécution des transitions qui composent son modèle, tandis que la vérification de propriétés temporelles s'appuie sur l'hypothèse que les transitions décrites sont valides pour donner des propriétés sur les états du système.

1.1.3 Validation d'un programme à partir d'un modèle formel

Nous nous intéressons, dans le cadre de cette thèse, à la validation d'un programme écrit manuellement à partir de spécifications des besoins. Ce processus s'oppose à la validation de programmes écrits par "génération de code" à partir d'un modèle, comme les logiciels embarqués dans le domaine de l'aérospatiale. En effet, une fois le modèle validé, c'est le générateur de code et le processeur sur lequel il est exécuté qui garantissent la conformité du programme généré avec le modèle ; ce n'est donc pas le cadre qui nous intéresse.

Valider un programme consiste à s'assurer du bon comportement de celui-ci. Comme pour la validation d'un modèle, le comportement du programme doit être comparé à une référence. Dans le cadre des travaux présentés dans cette thèse, c'est le modèle formel validé du système qui sert de référence.

Différentes techniques sont utilisées pour valider un programme par rapport à un modèle formel de celui-ci. La plus connue est le *test de conformité* [Bei95] qui consiste à simuler une séquence d'exécutions pertinente sur le modèle, à jouer cette séquence sur le programme et à comparer les résultats obtenus par le programme par rapport aux résultats attendus sur le modèle. Si les résultats sont conformes, le test est réussi ; dans le cas contraire, c'est un échec illustrant une non-conformité entre le programme et son modèle.

La *preuve de programme* est une autre technique qui s'intéresse à inférer les propriétés décrites dans le modèle à partir du programme source. Cette technique est plus spécifiquement employée en présence de langages de modélisation exprimés sous la forme d'annotations insérées dans les programmes, ce qui maximise la proximité entre le code et son modèle.

1.2 Contexte et problématique

Les travaux présentés dans cette thèse ont été réalisés au Laboratoire d'Informatique de l'Université de Franche-Comté (LIFC). Ce laboratoire, fondé en 1993 par le Pr. Michel

Tréhel, est actuellement dirigé par le Pr. Jacques Julliand depuis 1996. J'ai été accueilli en septembre 2003 au sein de l'équipe "Techniques Formelles et à Contraintes", régie par le Pr. Françoise Bellegarde. Mes travaux s'inscrivent dans le cadre de l'activité "Résolution de contraintes et génération de tests", menée par le Pr. Bruno Legeard, qui a co-encadré ces travaux avec Fabrice Bouquet.

Cette thèse s'insère dans le cadre de collaborations extérieures, à travers différents projets nationaux :

- l'ACI GECCOO (GEnération de Code Certifié pour les applications Orientées Objet), avec le Laboratoire de Recherche en Informatique (LRI) d'Orsay, le Laboratoire Logiciel Systèmes et Réseaux (LSR) de Grenoble, l'INRIA de Sophia-Antipolis et le Laboratoire Lorrain de Recherche en Informatique et en Automatique (LORIA) de Nancy. L'ACI GECCOO a également financé ces travaux.
- le projet INRIA CASSIS (Combinaison d'Approches pour la Sécurité des Systèmes InfiniS), bi-localisé entre le LIFC et le LORIA.
- le projet RNTL DANOCOPS (Détection Automatique de NOn-COnformité entre un Programme et sa Spécification), entre les partenaires industriels : THALES Systèmes Aéroportés et AxLog, et les partenaires universitaires : le LSR de Grenoble et le laboratoire Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S).

Le contexte scientifique de cette thèse et les problématiques qui y sont développées, sont liés aux savoir-faire de l'équipe et aux thématiques des différents projets dans lesquels je me suis impliqué. Le contexte scientifique général est celui de la validation de modèles formels par évaluation symbolique à contraintes. Cette recherche s'inscrit dans le cadre du projet BZ-Testing-Tools [BZT05], qui a permis à l'activité "Résolution de contraintes et génération de tests" d'acquérir ses lettres de noblesse avec la réalisation d'un environnement de validation de modèles B [Abr96] par animation et un générateur de tests aux limites à partir de ces modèles. Le champ d'application de ces technologies s'est vu élargi, à l'occasion de cette thèse, aux langages de modélisation orientés objet et plus précisément à JML, le langage de modélisation de Java.

1.2.1 Évaluation symbolique à contraintes

L'évaluation symbolique est une technique d'animation qui optimise le processus habituel d'animation. Elle consiste à utiliser des valeurs symboliques pour représenter les variables du système. Ces valeurs symboliques sont introduites au niveau des paramètres des opérations du système. Les valeurs symboliques sont gérées par des solveurs de contraintes, capables de garantir la cohérence de ces valeurs entre elles.

La programmation par contraintes consiste à programmer en utilisant des relations appelées *contraintes* entre les variables. Une contrainte s'applique à plusieurs variables et restreint les valeurs que celles-ci peuvent prendre simultanément. Trouver une solution consiste à affecter à chaque variable une valeur de son domaine de manière à ce que les contraintes soient satisfaites. On appelle *problème de satisfaction de contraintes* (CSP) un ensemble de variables, chacune ayant un domaine fini, sur lesquelles porte un certain

nombre de contraintes. La consistance de l'ensemble des contraintes présentes peut être assurée par différents mécanismes (consistance de nœud, d'arc, d'hyperarc, de bornes, de chemins, ou encore k-consistance). Les contraintes qui sont acquises peuvent, le cas échéant, être propagées pour réduire les domaines des variables mises en jeu. Lorsque deux contraintes sont détectées comme contradictoires (réduisant le domaine d'une ou plusieurs variables à l'ensemble vide), le CSP se révèle inconsistant. Suivant le mécanisme de consistance employé, la consistance ou l'inconsistance ne peut pas toujours être garantie au moment de l'acquisition d'une contrainte. Cette décision ne peut être obtenue qu'en parcourant l'ensemble des possibilités du CSP. L'automatisation de la résolution de CSP s'effectue dans un *solveur de contraintes*. Un solveur offre les possibilités suivantes : définition des domaines des variables, acquisition et propagation des contraintes, détection d'inconsistances, et instanciation.

L'animation symbolique applique les techniques de résolution de contraintes à l'animation d'un modèle. Les variables d'état d'un système en cours d'animation sont gérées par un solveur de contraintes. Le calcul d'une transition entre deux états du système est assimilé à un problème de satisfaction de contraintes entre les variables du système à l'état avant et celles à l'état après, sur lesquelles portent des contraintes issues d'une transition. L'introduction de valeurs symboliques est réalisée par l'intermédiaire des paramètres des opérations à activer. Lorsque ceux-ci sont laissés comme non-valués, ils sont associés à une nouvelle variable gérée par le solveur. Les variables d'état du système liées aux paramètres contraints sont alors impactées et deviennent elles-même contraintes.

Ces techniques sont implantées dans l'environnement BZ-Testing-Tools, développé au LIFC, pour permettre la validation d'un modèle B destiné à la génération automatique de tests. Elles constituent le point de départ de ces travaux de thèse.

La conception de modèles orientés objet vise à produire des implantations dans un langage de programmation orientée objet. Pour comprendre la différence entre la modélisation orientée objet et la modélisation classique comme en B il faut revenir aux principes de la programmation avec des objets.

1.2.2 La modélisation orientée objet

Les langages de programmation orientés objet se distinguent des langages "linéaires". La programmation orientée objet consiste à représenter le système considéré comme un ensemble d'objets caractérisés par des propriétés. Ainsi, un objet "mémoire de thèse" est composé –pour faire simple– d'un "titre", d'un "auteur", d'un certain "nombre de pages". Comme on peut le voir, certaines des propriétés peuvent être soit d'un type prédéfini (une chaîne de caractères pour le titre, un entier pour le nombre de pages), ou même d'un autre type d'objet. Par exemple, l'auteur du mémoire de thèse peut être vu comme un objet "personne", cette personne ayant elle-même un certain nombre de propriétés comme, par exemple, un nom, un âge, un poids, une taille et des parents sous la forme d'autres "personnes" différentes. Un objet qui existe est appelé une instance, chaque instance étant unique. Par exemple, l'exemplaire de thèse que le lecteur tient sous son regard peut être considéré comme une instance d'un "mémoire de thèse". Un autre lecteur disposant

d'un autre exemplaire aura en sa possession une autre instance. Mais il peut arriver que plusieurs lecteurs possèdent le même exemplaire de l'objet considéré. Dans ce cas, les modifications faites sur l'exemplaire commun par l'un des lecteurs (par ex. écritures dans les marges, tâches de café, leçon de coloriage intensif de jeunes enfants), affectent tous les lecteurs. Chaque objet est déclaré dans une *classe*. A chaque objet sont associées des opérations, nommées méthodes, qui peuvent s'appliquer sur l'objet. La classe "personne" peut ainsi contenir une méthode nommée "anniversaire()" qui permet d'augmenter l'âge de la personne d'une année.

Comme l'illustre le paragraphe précédent, réaliser un programme objet exige déjà une première phase obligatoire de modélisation : quels objets décrire ? quelles propriétés leur donner ? quelles méthodes leur associer ? Ceci donne une première étape de modélisation décrivant les informations statiques du système. C'est ce que proposent les diagrammes de classes UML (Unified Modeling Language) [RJB99] qui est la référence dans la modélisation de programmes objet. Malgré cela, il est souvent reproché à UML d'être peu formel. C'est ce qui a motivé la création du langage OCL (Object Constraint Language) [WK98] qui permet d'exprimer des propriétés formelles sur les classes contenues dans un diagramme de classes.

UML propose une approche généraliste, dans laquelle la modélisation est indépendante du langage d'implantation visé. C'est l'essence du Model-Driven Architecture (MDA) [OMG03] proposé par l'OMG, l'organisme ayant défini UML.

Plus récemment, sont apparus des langages de modélisation présentés sous la forme d'annotations embarquées dans le programme qu'elles modélisent. Ce principe a donné naissance au concept de Conception Par Contrat (ou Design By Contract) et a été défini par B. Meyer dans le langage Eiffel [Mey97] inspiré des travaux autour de Larch [GH93]. La modélisation par annotations est un procédé complémentaire à la modélisation de diagrammes de classes UML. En effet, ceux-ci visent à définir les classes du système, avec leurs attributs et leurs méthodes. L'emploi d'annotations permet de compléter la structuration précédemment définie, sans la remettre en cause, en décrivant les propriétés des classes et l'aspect fonctionnel de leurs méthodes.

C'est dans ce dernier cadre que se situe cette thèse, travaillant avec le langage de modélisation de Java [GJS00], le Java Modeling Language (JML) [LBR98, LBR99].

1.3 Contributions

Les contributions présentées dans ce mémoire sont ordonnées suivant la chaîne de validation des programmes Java à partir de modèles écrits en JML. Le point de départ est la représentation de modèles JML par des systèmes de contraintes. Celle-ci permet l'animation symbolique et ainsi, la validation des modèles JML. Les modèles validés sont ensuite utilisés pour la génération de tests. La démarche a été implantée dans un prototype nommé JML-Testing-Tools.

1.3.1 Représentation et interprétation symbolique

La première contribution est la représentation d'un programme Java/JML sous la forme d'un système de contraintes. Ainsi, nous interprétons le modèle de données Java en utilisant les structures logico-ensemblistes mises au point dans BZ-Testing-Tools. La traduction présentée garantit la préservation des propriétés initiales du système et exprime les contraintes d'intégrité existantes entre les instances de classes, notamment en présence d'héritage. Dans l'objectif d'animer le modèle, nous proposons l'expression des transitions du modèle JML sous la forme de pré- et postconditions. Pour compléter cette traduction, nous définissons les règles d'interprétation des systèmes de contraintes qui permettent de réaliser l'animation en elle-même. Celle-ci s'effectue en calculant le système de contraintes résultant de l'état avant du système et des contraintes extraites des spécifications de méthodes JML. Ces contributions ont été publiées dans [BDG05], où nous avons illustré la traduction d'un sous-ensemble de spécifications JML dans la notation B et dans [BDLU05b] où nous avons défini l'animation symbolique de modèles JML.

1.3.2 Génération de tests

La deuxième contribution est la mise au point de techniques de génération de tests, à partir des modèles JML qui permettent de confronter un programme sous test à sa modélisation. Dans ce cas, le choix des séquences de test et les stratégies mises en œuvre pour les générer sont abordés de manière originale. Ces méthodes de génération de tests sont basées à la fois sur l'expression des modèles de données Java, pour le calcul des données de test et sur l'expression des modèles JML sous la forme de contraintes, pour le calcul des séquences de test. Nous avons décrit dans [BDL06a] le mécanisme de génération automatique de tests aux limites à partir des modèles JML.

1.3.3 Validation de modèles

Enfin, nous présentons la validation de modèles par évaluation symbolique. Ceci est réalisé par des techniques semi-automatiques, permettant de calculer des séquences d'exécution du modèle, en s'appuyant sur les spécifications de méthodes JML pour s'assurer du bon comportement du modèle. Nous proposons également une approche automatique pour détecter des propriétés sur les modèles destinés à la génération de test. La vérification de ces propriétés génère des avertissements et des contre-exemples que l'utilisateur est libre de considérer ou non pour améliorer son modèle. Nous avons publié cette contribution dans [BDL05b], où nous avons expliqué le mécanisme d'animation symbolique de modèles logico-ensemblistes basées sur l'utilisation de systèmes de contraintes. Dans [BDL05a], nous avons décrit l'application de l'évaluation symbolique à la validation de modèles dans l'objectif de la génération de tests à partir de modèles.

1.3.4 Outils

La contribution technique présentée dans cette thèse est le développement d'un prototype implantant les règles de traduction et d'interprétation des modèles JML. Cet outil nommé JML-Testing-Tools est composé d'un animateur symbolique et d'un générateur de tests. L'animateur a été présenté dans [BDLU05a], et dans [BDL06b] dans le cadre de sessions outils. Nous avons expérimenté l'utilisation de l'animateur et du générateur de tests sur une étude de cas issue du monde industriel, décrivant un porte-monnaie électronique.

1.4 Plan du mémoire

Ce mémoire s'organise en 11 chapitres répartis en 3 parties. Nous présentons à présent la structure de ce document.

La **première partie** décrit le contexte de cette thèse. Elle contient les quatre chapitres suivants.

Le **chapitre 1** décrit le contexte des travaux présentés dans ce document et les problématiques abordées dans cette thèse.

Le **chapitre 2** présente l'état de l'art associé à ces travaux. L'état de l'art s'articule autour de la vérification et de la validation, principalement basée sur l'animation et le test fonctionnel.

Le **chapitre 3** décrit le contexte scientifique initial de cette thèse, à savoir la méthode BZ-Testing-Tools permettant la validation de modèles formels et la génération automatique de tests à partir de ceux-ci.

Le **chapitre 4** donne un aperçu des possibilités de modélisation offertes par les annotations JML. Ce chapitre présente la sémantique associée aux différentes clauses de modélisation de JML.

La **deuxième partie** contient les contributions apportées par cette thèse. Elle est composée des quatre chapitres suivants.

Le **chapitre 5** décrit la représentation logico-ensembliste choisie pour exprimer la structure des classes Java dans la notation intermédiaire de l'environnement BZ-Testing-Tools.

Le **chapitre 6** décrit la représentation choisie pour exprimer les spécifications JML de classes et de méthodes par des systèmes de contraintes, permettant ainsi l'animation des modèles JML à des fins de validation.

Le **chapitre 7** propose une approche de génération de tests aux limites inspirée de BZ-Testing-Tools et s'appuyant sur la définition et le calcul de tests pour des modélisations orientées objet décrites en JML.

Le **chapitre 8** présente l'évaluation symbolique des modèles JML dans le but d'aider le spécifieur à produire un modèle exploitable et efficace pour la génération de tests.

La **troisième partie** regroupe les réalisations et les expérimentations menées au cours de cette thèse.

Le **chapitre 9** décrit le prototype JML-TESTING-TOOLS qui permet l'animation symbolique des modèles JML et la génération automatique de tests à partir de ceux-ci.

Le **chapitre 10** rapporte une expérimentation menée sur une étude de cas, modélisant un porte-monnaie électronique pour la JavaCard. Cette modélisation est d'abord décrite puis nous présentons les cas de tests générés par notre approche et les problèmes soulevés par notre étude.

La **quatrième partie** clôture ce document en présentant les conclusions et en dressant les perspectives de ces travaux.

Chapitre 2

Etat de l'art

Sommaire

Les travaux présentés dans cette thèse sont liés à des activités relatives à la vérification de modèles, à la validation de modèles et au langage JML, qui est le langage de modélisation de Java, basé sur l'emploi d'annotations insérées à l'intérieur du programme. Cet état de l'art présente dans un premier temps un aperçu des techniques de vérification et de validation. Puis nous nous intéressons aux travaux liés à JML, avant de terminer par un aperçu des techniques de test, utilisées pour la validation de programmes, et leurs applications, en particulier à Java.

2.1 Techniques de vérification et de validation

Nous présentons ici un rappel des deux principales techniques de vérification qui sont la preuve (interactive ou automatique), ainsi que le model-checking (explicite ou symbolique). Nous poursuivons avec la présentation de différents travaux sur la validation de

modèles, soit par analyse symbolique, soit par animation. Les techniques de test, principalement utilisées pour la validation des programmes, seront décrites à la fin de cette partie.

2.1.1 Techniques de preuve

Les prouveurs de théorèmes sont basés sur le principe d'appliquer successivement des règles de déduction sur une formule écrite dans une logique donnée et permettent au final d'obtenir un verdict sur la vérité d'une formule.

Les approches par preuve de théorèmes vérifient des modèles, en produisant des formules, nommées *obligations de preuve*, qui sont ensuite injectées dans des prouveurs de théorèmes. Deux catégories de prouveurs se distinguent alors : les prouveurs interactifs et les prouveurs automatiques.

La preuve interactive

La preuve interactive est une procédure qui requiert l'expertise humaine pour mener à bien les déductions qui doivent s'effectuer sur la formule à prouver. On cite parmi les prouveurs interactifs les plus usités, COQ [CDT01, BC04], qui permet l'utilisation de tactiques destinées à automatiser une partie des déductions. Dans le même genre, HOL [GM93] traite de la logique d'ordre supérieur, et PVS [ORS92] intègre une partie interactive et une partie automatique, que nous décrivons à présent.

La preuve automatique

La démarche de preuve automatique se base sur la recherche de modèles de la formule à vérifier, par application automatique de règles de déductions. Le principal problème de cette approche concerne la divergence du prouveur qui peut, suite à l'application des règles de déduction, augmenter la taille de la formule ou boucler sur des schémas de déduction, empêchant la terminaison du calcul. Ce problème vient de la décidabilité de la théorie sous-jacente : si la théorie est décidable, il existe un algorithme qui permet de savoir si une formule écrite dans cette théorie est vraie ou fausse. Néanmoins, les conditions de décidabilité des théories sont extrêmement restreintes et on se contente la plupart du temps de semi-algorithmes qui peuvent ne pas terminer mais qui indiquent, lorsqu'ils terminent, un résultat correct. L'algorithme utilisé pour décider la satisfaisabilité ou la validité d'une formule est appelé "procédure de décision".

De nombreux prouveurs sont embarqués dans des outils de vérification. Par exemple, l'Atelier B [Cle01] intègre son propre prouveur automatique et interactif. ESC/Java, en charge de vérifier statiquement les programmes Java annotés en JML (cf. partie 2.2), utilise Simplify [DNS03]. Le projet INRIA Cassis a vu le développement du prouveur de théorèmes haRV^{ey} [DR03], qui a été utilisé pour la vérification de programmes [DR03] et la preuve de modèles B ensemblistes [CDD$^+$03]. La procédure de décision ICS [FORS01a] est intégrée dans l'environnement de preuve de PVS [ORS92].

Les techniques de preuve permettent de raisonner sur des systèmes potentiellement

infinis, ou paramétrés, car cette technique ne s'intéresse pas à énumérer de manière exhaustive tous les états du système. Ce fonctionnement est propre aux model-checkers.

2.1.2 Model-checking explicite vs. Model-checking symbolique

Le model-checking est une technique qui s'appuie sur le calcul complet du graphe d'atteignabilité d'un système. On distingue deux types de model-checking : le model-checking explicite [CGP99], qui énumère tous les états et toutes les valeurs des paramètres pour les transitions entre les états, et le model-checking symbolique [McM92], plus récent, qui considère des valeurs symboliques pour les paramètres des transitions. Il existe de nombreux model-checkers explicites, tels que SPIN [Hol91] utilisant des modèles décrits en Promela, ou encore ProB [LB03] destiné aux modèles B [Abr96].

Le model-checking explicite souffre du problème de l'explosion combinatoire de la taille du graphe des états atteignables. Le model-checking symbolique, initié par les travaux de K. MacMillan [McM92], retarde l'explosion combinatoire en représentant les ensembles des configurations accessibles de manière symbolique, par des systèmes de contraintes. Les travaux de B. Parreaux [Par00] menés au LIFC ont repris ces idées pour les appliquer à la vérification de propriétés PLTL.

2.1.3 Techniques de validation de modèle

La validation de modèles est la plupart du temps instrumentée dans des outils nommés "CASE tools" (Computer-Aided Software Engineering tools – outil de génie logiciel assistés par ordinateur), qui fournissent des environnements présentant diverses fonctionnalités, comme la vérification de la cohérence du modèle, et la génération de code exécutable.

La validation d'un modèle formel est une étape importante, permettant de s'assurer du bon comportement du modèle, en particulier lorsque celui-ci vise à être utilisé pour la génération de tests ou la génération de code. Une première approche permet de considérer le model-checking comme une possibilité de validation d'un modèle, à travers le mécanisme de vérification des propriétés temporelles, de sûreté ou de vivacité. Nous divisons les techniques de validation de modèles en deux catégories : les techniques basées sur l'analyse symbolique, et les techniques basées sur l'animation. Nous décrivons à présent les travaux relatifs à ces catégories. Pour chacune d'elles, nous présentons quelques outils représentatifs.

Validation par analyse symbolique

Nous nous intéressons ici aux travaux ayant trait à la validation de modèles par analyse symbolique. Nous présentons dans cette catégorie l'analyseur Alloy, dont le langage d'annotations n'est pas sans rappeler JML et qui est donc proche des travaux présentés dans ce mémoire, puis nous verrons l'outil SAL qui est un environnement complet destiné à la validation de modèles par analyse symbolique.

Alloy Alloy [Jac02] est un langage déclaratif permettant la description de modélisations formelles basées sur les ensembles et les relations. Le modèle est exprimé en déclarant des

structures décrivant le modèle de données, des prédicats décrivant des conditions, des invariants, des assertions et des transitions.

Un analyseur spécifique pour Alloy, nommé Alcoa (Alloy Constraint Analyzer) [JSS00] a été mis au point pour réaliser des vérifications sur les modèles Alloy. Etant donné un modèle et une assertion, Alloy est capable de trouver une *instance* du modèle qui satisfait l'assertion. Alloy répond à deux problèmes : la faiblesse des contraintes données dans le modèle et, à l'inverse, la trop grande force de ces contraintes. Le premier problème se résout en établissant la préservation des invariants par une opération, le raffinement d'une opération par une autre, ou l'implication d'un invariant par un autre. Le second problème se résout en exerçant les invariants ou les opérations pour essayer de trouver des états ou des transitions satisfaisants les contraintes.

Alcoa fonctionne sur la résolution de contraintes sur domaines finis en limitant le nombre d'éléments dans chaque type primitif. Alcoa produit en sortie une instance ou un message indiquant qu'aucune instance ne peut satisfaire les contraintes imposées. Lorsqu'une assertion est vérifiée, la négation de l'assertion est considérée et Alcoa recherche une instance satisfaisante, qui présente un contre-exemple indiquant que le théorème n'est pas valide. En vérifiant une opération, une instance illustre la consistance. La résolution de contraintes est effectuée par des solveurs SAT, notamment SATO [Zha97] et RelSAT [BS97], après traduction du modèle analysé en une formule booléenne.

Un langage, nommé Alloy Annotation Language (AAL) [KMJ02], a été mis ou point pour permettre d'annoter les programmes Java et permettre ainsi d'utiliser les capacités d'Alloy, et des outils dédiés, pour l'analyse des programmes Java. AAL est très similaire à JML.

SAL Symbolic Analysis Laboratory (SAL) [BGL+00] est un environnement utilisé pour vérifier, par analyse symbolique, des propriétés sur les programmes concurrents. SAL est un langage intermédiaire développé en collaboration entre Stanford, Berkeley et Verimag pour la spécification de systèmes concurrents de manière compositionnelle. L'environnement SAL est une sur-couche à PVS qui propose des outils pour l'abstraction, la génération d'invariants, l'analyse de programmes comme le "slicing" [DH99], et le model-checking. SAL transforme une modélisation SAL en différents langages cibles, visant chacun une utilisation particulière. Ainsi, un traducteur vers PVS [ORS92] permet d'utiliser les capacités de preuve de théorèmes de ce dernier, un traducteur vers SMV [K.L92] permet de réaliser le model-checking du modèle, et un traducteur vers Java permet d'animer celui-ci, embarquant la vérification des propriétés du programme dans un mécanisme d'exceptions levées lors de l'exécution.

Après avoir illustré les principes de validation de modèle par analyse statique à travers les possibilités offertes par deux outils représentatifs de cette catégorie, nous présentons les techniques de validation des modèles par animation.

Validation de modèles par animation

L'animation de modèles formels permet la détection d'erreurs dans le modèle à un niveau moindre que la preuve, pour ce qui concerne la cohérence d'un modèle. En effet,

tout comme pour le test, l'animation ne peut que montrer la présence d'erreurs, et non leur absence [MS01].

L'animation des modèles est surtout liée à certains langages comme Z, dont l'expressivité présente un challenge à l'animation. On trouve dans la littérature de nombreux travaux sur l'animation de tels modèles qui visent surtout à décrire le fonctionnement de l'animation des différentes notations, comme par exemple [HOS97, Jia95, MS03] pour Z ou [GS00] pour UML/OCL, mais peu de travaux se consacrent à l'utilisation de l'animation comme un moyen à part entière de valider un modèle.

Dans cet esprit, [MS01] présente l'utilisation de Possum [HST97] un animateur de modèles Sum [KKTW], une extension modulaire de Z, pour réaliser l'animation "systématique" des modèles. Par systématique, les auteurs entendent créer des cas d'animation du modèle qui sont planifiés, documentés et maintenus. Ainsi, des cas d'animation sont mis au point pour exercer le modèle et ces cas peuvent être rejoués en cas de modification du modèle. Dans cette approche, l'animation s'apparente à un test de la modélisation et rejouer les séquences d'animation peut être assimilé à des tests de non-régression lors des évolutions successives du modèle.

SALT Les travaux présentés dans [PK04] proposent la vérification de différentes propriétés sur des modèles formels écrits en SALT (Specification and Abstraction Language for Testing). Ces travaux s'intéressent à l'analyse de la consistance et de la complétude de ces modèles. La consistance se définit comme un comportement unique pour chaque combinaison de paramètres d'entrée et d'états du système ; la complétude est identifiée si chaque comportement du système est défini pour toutes les combinaisons de paramètres d'entrées et d'états du système.

Ainsi, les analyses effectuées tendent à détecter des contradictions dans les conditions des gardes, des chevauchements des gardes ayant des conséquences contradictoires, des conditions de garde ne couvrant pas toutes les possibilités de paramètres d'entrée et d'états du système, les valeurs manquantes, les gardes redondantes, et l'atteignabilité d'un état du système satisfaisant les gardes des opérations.

Model-checking borné Le model-checking borné [CBRZ01] (Bounded-Model Checking – BMC) est une technique d'optimisation du model-checking symbolique, qui se base sur l'utilisation de solveur SAT propositionnels plutôt que sur la représentation symbolique par des BDDs, comme c'est le cas en model-checking symbolique. Le model-checking borné prend en paramètre une limite qui indique la profondeur de recherche maximale à considérer. Les contraintes relatives aux différentes transitions parcourues lors de l'exploration du graphe d'état symbolique sont accumulées et déchargées à un solveur SAT qui est en charge de déterminer la satisfaisabilité. En pratique, le model-checking borné est utilisé pour la vérification de propriétés de sécurité [SSS00], en recherchant les contre-exemples d'une propriété à une profondeur donnée. En cas de violation de la propriété, l'instanciation du système de contraintes fournit une trace d'exécution.

On notera au passage que des travaux récents ont été menés [HdR04] pour utiliser cette technique dans le cadre de la génération de cas de test. L'objectif de test est décrit comme une propriété à atteindre, et l'application du BMC permet de trouver une trace

d'exécution menant à l'objectif de test, ce qui constitue ainsi le cas de test.

2.1.4 Bilan

Dans le cadre de la validation de modèles, diverses techniques sont employées, s'appuyant sur l'animation du modèle ou la vérification de propriétés. Ces techniques peuvent ainsi être assimilées à des mécanismes de preuve, de model-checking ou de résolution de contraintes.

Les travaux présentés dans cette thèse se focalisent sur l'animation de modèles JML, en utilisant des systèmes de contraintes. Ainsi, l'animation offre la possibilité de valider le modèle formel pour s'assurer qu'il respecte bien les exigences requises pour la génération de tests. L'emploi de méthodes symboliques, reposant sur des solveurs de contraintes, est une approche intéressante, et fréquemment employée (Alloy, SAL, Model-checking borné) permettant de retarder l'explosion du nombre d'états manipulés.

2.2 Travaux autour de JML

Dans cette partie de l'état de l'art, nous nous intéressons aux outils et travaux liés à JML [LBR98, LBR99, LBR02] et ayant trait aux problèmes auxquels répondent les travaux présentés dans ce mémoire. Nous considérons les travaux liés à la génération de tests à part ; ceux-ci seront décrits de manière générale dans la partie 2.3.

Nous présentons ici les deux applications principales de JML, à travers les outils qui lui sont dédiés. Ces deux applications sont de considérer les annotations JML comme un oracle pour l'exécution de tests sur un programme Java, ou pour aider à la vérification du code Java. Nous commençons par présenter les idées qui sont à l'origine de JML, accompagnées des outils fournis dans la distribution de JML [JML05]. Nous donnerons ensuite un aperçu des travaux menés sur la vérification de la conformité des programmes Java par rapport aux modèles JML.

2.2.1 Un oracle pour le test de programmes Java

L'idée d'origine de JML est de fournir la possibilité de définir des contrats pour l'exécution des méthodes JML [LCC+03]. Ainsi, la distribution officielle de JML fournit principalement deux outils ; le premier, le Runtime Assertion Checker, sert de monitoring au code Java et le second, JMLUnit, s'appuyant sur le précédent, est dédié à l'intégration de JML dans le test unitaire de méthodes Java.

Runtime Assertion Checker Le Runtime Assertion Checker [CL02a] de JML est un outil permettant de vérifier les annotations JML à la volée lors de l'exécution du code Java. Pour ce faire, les annotations JML sont traduites en instructions Java effectuant la vérification des prédicats contenus dans le modèle JML. La proximité entre ces deux langages est la clé d'une telle démarche ; en effet, la syntaxe des prédicats JML est similaire à celle des prédicats Java et les mots-clés JML spécifiques (par exemple les quantificateurs) sont traduits par des structures Java adéquat (dans ce cas, une boucle). Néanmoins toutes

les annotations ne sont pas traduisibles et certaines constructions JML ne peuvent être vérifiés par ce moyen (par exemple les quantifications sur les objets).

Le code Java d'origine est ainsi enrichi par la vérification des annotations JML ce qui a pour effet d'augmenter la taille du byte-code engendré et de ralentir le temps d'exécution du programme. Au cours de l'exécution, si une assertion JML n'est pas vérifiée, une exception spécifique est déclenchée signalant le type d'annotation qui n'a pas été vérifiée (invariant, précondition, etc.) et l'état visible du système au moment de l'exécution.

L'utilisation du Runtime Assertion Checker confère un moyen direct et efficace de s'assurer que les assertions JML ne sont pas violées durant une exécution. Ce mécanisme fournit donc un moyen de comparer les résultats calculés par le modèle et les résultats calculés par le programme, il peut donc servir d'oracle pour donner le verdict de l'exécution de séquences de tests. Nous décrivons à présent l'intégration des possibilités offertes par le RAC dans un environnement dédié à l'exécution de tests.

JMLUnit JMLUnit est la combinaison de JML et de JUnit [GB98], un outils d'exécution de tests unitaires pour Java. Le plus gros problème de JUnit est qu'il requiert de l'utilisateur qu'il fournisse à la fois les séquences de tests, les données de tests, et l'oracle pour calculer le verdict des tests exécutés. JUnit présente alors, suite à l'exécution des tests un récapitulatif des tests ayant réussi et ceux ayant échoué.

La combinaison de JUnit et du RAC a été présentée dans [CL02b] et intitulée JMLUnit. Il s'agit d'un environnement de tests à la JUnit mais intégrant un oracle automatique, issu des assertions JML intégrées grâce au Runtime Assertion Checker.

Ces deux outils feront l'objet d'une présentation plus spécifique suite à la présentation de JML au chapitre 4. Nous nous intéressons à présent à une utilisation connexe de JML qui est la vérification de programmes.

2.2.2 Vérification de programmes

JML fournit un cadre formel à la description du comportement des interfaces et des classes Java [LBR02]. La vérification des programmes Java s'appuie sur l'utilisation de JML comme renfort au code. Les outils présentés dans cette partie sont dédiés à la détection statique d'erreurs d'exécution dans les programmes Java/JML, telles que les déréférencements de pointeurs null, les divisions par zéro, ou les dépassements d'indices dans les tableaux.

ESC/Java2 ESC/Java2 (Extended Static Checker) [CK04] est la suite des travaux de ESC/Java [FLL+02] initialement menés par Compaq[2]. Il s'agit d'une approche pragmatique et simple, qui vise à aider à la détection des erreurs précédemment citées. ESC/Java2, repris par KindSoftware[3], intègre le support des annotations JML et vérifie en plus la cohérence des annotations et du programme. Néanmoins, ESC/Java2 n'est pas un outil de preuve, il s'appuie sur des techniques de réfutation qui visent à trouver un contre-exemple.

[2] `http://research.compaq.com/SRC/esc`

[3] `http://secure.ucd.ie/products/opensource/ESCJava2/`

A partir d'un programme annoté, ESC/Java2 génère des conditions de vérification qui représentent des conditions garantissant l'absence des erreurs "classiques" décrites précédemment. ESC/Java2 génère une condition de vérification par type d'erreur. Celles-ci sont déchargées dans le prouveur Simplify [DNS03]. Celui-ci décide alors si la condition de vérification est valide ou invalide et produit dans le second cas un contre-exemple. Celui-ci est ensuite analysé et rendu à l'utilisateur pour produire un avertissement relatif au type d'erreur considéré. A l'inverse, si la condition de vérification est valide, alors le programme est exempt du type d'erreur considéré. Si ESC/Java2 n'est ni exact ("sound") ni complet, en pratique, il trouve beaucoup d'erreurs [CK04].

JACK JACK (Java Applet Correctness Kit) [BRL03] a été initialement développé au laboratoire de recherche de Gemplus[4]. Il est désormais développé par l'INRIA Sophia Antipolis.

Il s'agit d'un environnement pour la vérification de programmes Java et JavaCard annotés avec JML, s'exécutant sous l'environnement de développement Eclipse[5]. JACK implante un calcul de plus faible précondition qui génère des obligations de preuve à partir des sources Java. L'avantage de JACK est qu'il permet de générer des obligations de preuve pour différents prouveurs aussi bien automatiques qu'interactifs, tels que le prouveur de l'AtelierB [Cle01], Simplify [DNS03], l'assistant de preuve Coq [CDT01] ou PVS [ORS92].

JACK génère des lemmes qui garantissent que le programme implante bien le modèle JML donné. Pour plus de clarté, JACK propose aux utilisateurs de visualiser ces lemmes dans la syntaxe de Java. Ces lemmes sont exprimés dans un langage interne, nommé JPOL (Java/JML Proof Obligation Language), mélange de JML, Java et de la théorie des ensembles. JPOL utilise une théorie "background" minimale (pour augmenter sa décidabilité) axiomatisant les références, les instances, les types et sous-types, et les tableaux. La connexion entre les lemmes et les différents prouveurs est assurée par des plug-in Eclipse réalisant la traduction de JPOL vers les formats d'entrée des différents prouveurs.

JACK a été expérimenté dans le domaine des applications de téléphonie mobile et carte à puce [PBB+04].

Krakatoa Krakatoa [MPMU03] est un générateur d'obligations de preuve pour le prouveur interactif Coq [CDT01]. Krakatoa vise à vérifier l'exactitude des implantations des méthodes vis-à-vis de leurs modèles JML. Les obligations de preuve générées par Krakatoa visent à s'assurer que les invariants de classe et les préconditions sont vérifiés au début d'une méthode et que les invariants et les postconditions sont vérifiés à la fin de la méthode.

Pour effectuer la vérification, Krakatoa s'appuie sur l'outil WHY [Fil03] pour générer les obligations de preuve dans le formalisme de Coq, Simplify ou haRV[ey]. WHY utilise son propre langage de description qui se présente sous la forme d'un langage fonctionnel annoté avec des préconditions, postconditions et des invariants de boucle. Krakatoa compile les

[4] `http://www.gemplus.com`
[5] `http://www.eclipse.org`

programmes Java en WHY qui se charge ensuite de générer les obligations de preuve dans les formalismes cibles.

Model-checking Les derniers travaux que nous présentons dans la catégorie de la vérification de programmes Java/JML est l'approche de model-checking décrite dans [RRDH04]. Les auteurs présentent l'utilisation du model-checker Bogor [RD03] pour la vérification de modèles JML. Bogor est un environnement de model-checking extensible et modulaire pouvant être adapté aux spécificités de différents domaines. Bogor utilise son propre langage de description BIR qui permet d'introduire des types de données abstraits qui formeront les états des machines abstraites produites par le model-checker. Bogor autorise ainsi l'expression de notations objet et permet une bonne couverture des opérateurs JML, de part sa représentation interne des structures de données. Les modèles JML et le code Java correspondant aux méthodes sont traduits au format BIR pour permettre les vérifications : des pré- et postconditions des méthodes, des invariants, des attributs modifiés. Cette approche permet également de gérer le mécanisme de concurrence.

2.2.3 Bilan

Les approches dédiées à JML s'articulent principalement autour de la vérification du programme par rapport à son modèle, que ce soit dans le contexte particulier d'une séquence d'exécutions, avec les outils dédiés au test, ou dans le cadre plus général de la preuve, mais plus limité du point de vue des structures supportées. Notre approche diffère en ce sens qu'elle s'intéresse à JML comme un langage de modélisation à part entière. Ainsi, le code des méthodes Java n'intervient pas dans le processus d'animation que nous proposons, et seule la modélisation JML est utilisée pour calculer les transitions entre les états. Néanmoins, nous nous intéressons à la conformité entre le programme et son modèle. Pour cela, nous mettons en œuvre un processus de génération de tests qui va tenter, par la création d'une séquence d'exécution de mettre en défaut le programme Java par rapport au modèle JML qui lui est associé. Dans cet objectif, nous utilisons le Runtime Assertion Checker pour donner le verdict de (non-)conformité.

2.3 Génération de tests

Le principe du test est de stimuler un système avec différentes opérations, en fournissant différents paramètres et en vérifiant le résultat obtenu. Tout l'intérêt du test est de choisir les bonnes séquences d'opérations et les paramètres qui sont les plus pertinents.

L'objectif du test est de trouver des erreurs sur le système. Pour ce faire, les séquences de test sont passées sur le système sous test, et les résultats rendus par celui-ci sont ensuite observés. Si ceux-ci sont conformes aux résultats attendus, appelés *l'oracle*, le test réussit. Dans le cas contraire, le test échoue. On parle alors de verdict de test.

Les techniques de test ne sont pas exhaustives et l'absence d'échecs lors du passage des tests n'est en aucun cas une garantie de bon fonctionnement du système. Néanmoins, suivant les critères utilisés pour la génération des tests, et suivant la couverture fournie par les tests, un système ainsi *validé* peut acquérir un certain niveau de confiance auprès

de ses utilisateurs.

On distingue différentes techniques de génération de tests, présentant différents niveaux d'automatisation (manuelle, semi-automatique, automatique), et différents niveaux de connaissance du programme qui est testé (complète, partielle, inconnue). Nous ne nous intéressons dans ce mémoire qu'aux techniques automatisées visant à dériver des tests à partir d'un programme ou d'un modèle de ce programme.

Cette partie détaille les techniques de génération de tests existantes qui sont en rapport avec les techniques citées et employées dans ces travaux de thèse. Nous commençons par l'approche dite "basée sur le modèle" (ou *model-based testing* – MBT). Pour celle-ci, nous présentons des techniques ayant attrait à la construction de systèmes de transitions étiquetés, de machines à états finis et résolution de contraintes. Si cette dernière technique est l'objet des travaux présentés dans ce mémoire, les autres techniques ne sont pas pour autant sans intérêt car elles pourraient également s'appliquer à Java/JML.

Parmi ces autres techniques, ne s'appuyant pas nécessairement sur l'établissement d'un modèle pour générer les séquences de test, nous réduisons le champ de nos investigations aux approches visant à produire des cas de test pour les programmes Java.

2.3.1 Model-Based Testing

Le principe de la génération de tests à partir de modèles est d'utiliser un modèle formel décrivant le système à tester de manière fonctionnelle, puis d'animer celui-ci pour générer des cas de tests avec oracle. Ces cas de tests sont ensuite confrontés aux exécutions réelles du programme.

La plupart du temps, la modélisation sur laquelle se base la génération de tests est une abstraction du système qui doit être testé. Ainsi, des séquences de test dites *abstraites* sont produites, présentant le même niveau d'abstraction que le modèle d'origine. Ces séquences abstraites doivent ensuite être *concrétisées* pour pouvoir passer sur le système sous test (System Under Test – SUT).

La génération des séquences de test est le cœur du problème. La représentation fonctionnelle qui est faite du système et le fait que les séquences de test sont dérivées à partir du modèle confèrent une connotation de test *boîte noire* [Bei95] pour laquelle l'implantation n'est pas connue.

Il existe différents moyens de dériver des tests à partir des modèles. Ceux-ci dépendent des types de modélisation du système et des stratégies employées pour calculer les tests. Nous allons aborder dans le reste de cette partie les différentes techniques utilisées pour générer des cas de test. Ces techniques sont classées suivant les théories sous-jacentes : la génération de machines à états finis, les systèmes de transitions étiquetés, et la résolution de contraintes. Nous décrivons à présent ces concepts et nous les illustrons avec les outils qui ont été associés.

Machines à états finis

La génération de tests à partir de machines à états finis (Finite State Machines – FSM) [Cho78, LY96] consiste à représenter chaque état du système sous test par un nœud de

la FSM, et chaque opération du système par un arc. Le processus de génération de tests consiste ensuite à parcourir l'automate ainsi produit pour générer des séquences de tests. Pour ce faire, différentes techniques de parcours existent, la plus célèbre étant celle du *"Rural Chinese Postman"*, qui consiste à activer chaque action du système en trouvant le chemin le plus efficace [Rob99].

Nous illustrons le fonctionnement des machines à états finis par l'intermédiaire de deux outils : AsmL Test Tool et SpecExplorer.

AsmL Test Tool AsmL Test Tool [BGS+03] est un outil de génération de tests basé sur la théorie des machines à états abstraits (Abstract State Machines – ASM), mises au point par Y. Gurevich [BG94] et qui est de nos jours utilisée pour définir la sémantique des langages de programmation, en les représentant par des états abstraits sur lesquels peuvent s'exercer des transformations basiques faisant évoluer les états. AsmL (Abstract State Machine Language) est le langage de description des ASM, supervisé par le groupe Foundations of Software Engineering (FSE) de Microsoft [Gur05] dirigé par Y. Gurevich.

Le processus de génération de tests à partir d'AsmL [GGSV01] se déroule en deux étapes. La première étape consiste à traduire l'ASM du modèle en une machine à états finis. La seconde étape consiste ensuite à appliquer des algorithmes basés sur les FSM pour produire les suites de tests. La principale difficulté dans ce processus est la traduction d'une ASM en FSM [GGSV02]. En effet, la plupart du temps, les ASM présentent un nombre infinis d'états atteignables. Ceux-ci sont alors regroupés par classes d'équivalence. Ces dernières doivent être en nombre fini, et suffisamment grand pour produire un objectif de test significatif. Basiquement, les ensembles d'états représentant les classes d'équivalence deviennent les états de la FSM.

La FSM ainsi calculée représente le comportement du système basé sur les appels de méthodes. Les appels de méthodes représentent ainsi des actions d'entrée et les résultats correspondent aux actions de sortie.

SpecExplorer SpecExplorer [CGN+05] est un outil de génération de tests pour C# [Tro01] par exploration du modèle du programme décrit en AsmL ou en Spec# [MBS04]. Ce dernier définit les états abstraits et les transitions d'une ASM, à partir de laquelle les tests sont générés. Les transitions entre les états sont les invocations des méthodes du modèle qui satisfont leurs préconditions.

SpecExplorer parcourt les états de la machine et les transitions à l'aide de techniques classiques héritées du model-checking. Pour dériver les cas de test, il est nécessaire de préciser les états d'acceptation de la FSM, qui permettent de déterminer la fin du calcul des séquences de test. La sélection des données de test s'effectue suivant différents modes : par défaut (des valeurs prédéfinies sont choisies pour les paramètres), par type (des valeurs redéfinissant les valeurs possibles pour un type donné), par paramètre (des valeurs données par des expressions basées sur l'état du système), ou par méthode (en créant des tuples d'expressions représentant les paramètres associés à la méthode activée).

Une séquence de test produite à partir d'un automate est un automate décrivant la séquence de test. Cet automate présente les propriétés suivantes. (*i*) Il peut faire référence à des variables spécifiques au modèle. (*ii*) Ses transitions se composent de transitions de

test et d'actions de test (respectivement les transitions du systèmes et les appels à des méthodes d'observation). (*iii*) Un état d'acceptation est atteignable depuis n'importe quel état de cet automate. (*iv*) Chaque état est soit actif, s'il est atteint par une transition de test, soit passif, s'il est atteint par une action d'observation de test. (*v*) Il n'y a pas de boucles.

Cette approche a été employée dans différents cas de figure, tels que la génération de tests pour interfaces graphiques [PFTV05].

Nous nous intéressons désormais à la génération de tests à partir de systèmes de transitions étiquetés qui sont une généralisation des FSM, comme nous allons le voir à présent.

Systèmes de transitions étiquetés

La génération de tests à partir de systèmes de transitions étiquetés (Labeled Transition Systems – LTS) consiste à représenter les actions du système comme des relations de transition entre deux états. La différence principale entre un système de transition étiqueté et une machine à états finis est que le LTS peut avoir un ensemble infinis d'états ou de transitions, tandis qu'une FSM est limitée à un ensemble fini d'états et un alphabet d'entrée fini.

Nous illustrons le principe de génération de tests à partir de systèmes de transitions étiquetés à travers les outils TGV et STG.

TGV L'outil Test Generation with Verification technology (TGV) [Jér02] est développé en commun par l'IRISA de Rennes et Verimag à Grenoble. Il s'agit d'un générateur de tests qui implante la relation d'implantation **ioco** [Tre96], qui considère une spécification décrite par un LTS, et l'implantation modélisée par un IOTS (Input Output Transition System). Une relation de conformité est décrite entre ces deux systèmes de transition, ce qui permet de définir si le test réussit ou échoue. TGV se base sur un IOLTS (Input-Output Labeled Transition System), c'est-à-dire, un LTS partitionné en trois ensembles, décrivant les stimuli, les observations et les actions internes.

TGV admet en entrée un modèle et un objectif de test, définis tous deux par des IOLTS. Les verdicts possibles sont *pass*, *fail*, *inconclusive*. Un objectif de test est défini par un IOLTS déterministe et complet, présentant deux états "pièges" (finaux) *Accept* et *Refuse*. Le produit synchrone des deux IOLTS est calculé et les cas de test sont extraits en sélectionnant les comportements acceptés, c'est-à-dire, les traces d'exécutions qui mènent à l'état *Accept*.

TGV est disponible dans le CAESAR/ALDEBARAN Development Package (CADP) [JHA+96]. Il accepte les langages de spécifications LOTOS (à travers CADP), SDL (soit à travers le simulateur d'ObjectGéode SDL [GJK99] ou en utilisant l'outil TestComposer [SEG00]), UMLAUT [JHGP99] et IF (en utilisant le simulateur du compilateur IF [BFG+99])

STG L'outil Symbolic Test Generator [CJRZ02] a été développé à l'IRISA/INRIA de Rennes. STG permet la génération de tests, ainsi que l'exécution des tests de manière

symbolique. Il prend en entrée un IOSTS (Input-Output Symbolic Transition System – Système de Transition Symbolique à Entrées-Sorties) [HM00], et un objectif de test à partir desquels il produit un cas de test symbolique. STG est une évolution de TGV utilisant un mécanisme symbolique, dont le principe de fonctionnement est similaire.

L'objectif de test est également décrit sous la forme d'un modèle IOSTS. Il peut faire référence aux paramètres et aux variables du système modélisé pour sélectionner le comportement ciblé. Un objectif de test présente deux nœuds *Accept* et *Reject*, le premier explicitant la partie du système que l'on souhaite tester, le second, décrivant la partie du système qui ne présente pas d'intérêt.

Un cas de test symbolique consiste en un programme réactif qui couvre tous les comportements du modèle qui est visé par l'objectif de test. Pour être exécuté, un cas de test symbolique abstrait est traduit en un programme concret qui est lié à l'implantation. Le programme résultant est ensuite exécuté et il amène aux trois résultats : succès, échec, ou inconclusif.

STG s'appuie sur le model-checker HyTech [HHWT95] et sur le prouveur de théorèmes PVS [ORR+96] pour simplifier les tests générés en élaguant les parties inatteignables dues aux gardes inconsistantes [RdBJ00]. De plus, STG a été couplé à PVS pour combiner test et vérification [Rus02]. STG a été mis en œuvre pour tester des versions simples du CEPS (Common Electronic Purse Specification) [CEP01], et des cartes à puces 3GPP (Third Generation Partnership Program) [CJRZ01].

Nous terminons notre tour d'horizon des techniques de génération de tests par les techniques basées sur la résolution de contraintes.

Résolution de contraintes

Les techniques de résolution de contraintes présentent l'avantage de retarder le plus possible l'énumération des états du système en utilisant une représentation symbolique des états et des transitions entre les états. Nous illustrons les principes de la génération de tests par résolution de contraintes à travers les outils AGATHA et GATeL.

AGATHA L'outil AGATHA (Atelier de Génération Automatique de Tests Holistiques pour Automates) [BFG+03] est développé au CEA[6], dans l'équipe LIST[7].

AGATHA est dédié à la validation et à la génération automatique de tests. Cet outil se base sur l'exécution symbolique de systèmes constitués d'automates concurrents. Il accepte des spécifications décrites en Estelle [Bul89], SDL [EhS97], Statecharts UML [vdB01], ou Statecharts Statemate [HN96]. AGATHA génère un arbre d'exécution complet qui couvre l'ensemble des comportements du système analysé et fournit pour chacun des comportements un test numérique. Le calcul du graphe d'exécution symbolique du système est réalisée grâce à l'utilisation de différentes techniques qui sont :

- L'exécution symbolique, qui consiste à ne pas considérer les paramètres d'entrée comme évalués mais comme des valeurs symboliques sur lesquelles portent des contraintes issues des gardes des transitions du système.

[6] Commissariat à l'Energie Atomique

[7] Laboratoire d'Intégration des Systèmes et des Technologies

- Les procédures de réduction, qui visent à simplifier à la volée les expressions symboliques, grâce à l'emploi de l'outil CafeObj [Caf05].
- La composition, qui s'intéresse à fusionner différents automates, en synchronisant les transitions qui ont rendez-vous ou en les remplaçant par des transitions équivalentes dans lesquelles les paramètres de communication ont été retirés.
- Les solveurs de contraintes, qui sont employés pour obtenir les valeurs numériques des paramètres du système par résolution des contraintes de chemin accumulées lors de l'exécution symbolique. C'est le solveur Omega [KMP+96] qui est en charge de cette tâche.

AGATHA a été utilisé dans des environnements d'outils industriels comme Object-Géode [Tel99], Statemate [IL005] ou Objecteering [Sof99], et a été employé dans divers domaines de systèmes embarqués tels que l'aérospatial ou l'automobile [PGF+00, GLL99].

GATeL GATel [MA00] est un outil de génération de tests également développé au CEA dans l'équipe LIST qui s'appuie sur des spécifications décrites dans le langage Lustre [HCRP91]. GATeL se base sur une représentation symbolique par systèmes de contraintes des états du système et construit les cas de test par chaînage arrière à partir de l'état cible. L'état cible est décrit à l'aide de contraintes qui peuvent contenir aussi bien des invariants que d'autres contraintes sur le passé, telles qu'il est possible de les exprimer en Lustre. Ces dernières décrivent un objectif de test, puisqu'elles permettent de définir un prédicat caractérisant les séquences de test jugées pertinentes. Lors de la génération des séquences de test, GATeL recherche une séquence qui satisfasse à la fois l'invariant et l'objectif de test.

Cet outil offre la possibilité de réaliser des "séparation de domaines" (domain-splitting) qui visent à remplacer les domaines des variables par des sous-domaines définis comme les des cas particuliers des domaines d'origine. Cette stratégie a pour objectif de multiplier les cas de test en considérant des couvertures de données plus fines. GATeL est implanté en Prolog, et basé sur le package ECLiPSE [ECL05].

2.3.2 La génération de tests pour Java

La génération de tests pour Java se décompose en deux catégories. D'un côté, on trouve les concepts et les outils qui visent à générer des structures de données objet complètes et complexes, qui se focalisent sur du test unitaire. Ceux-ci supposent la possibilité de construire directement à moindre coût les objets en ayant accès à leurs attributs (soit à travers des méthodes spécifiques, soit de par leur visibilité). D'autre part, on trouve des outils qui s'intéressent plus à construire des séquences d'exécutions qui sont supposées faire évoluer un nombre fini d'objets dans le but de les pousser à l'erreur. Dans les deux cas, le modèle JML n'est utilisé qu'avec parcimonie, pour fournir l'oracle ou pour filtrer les valeurs fournies en précondition.

Nous décrivons en premier lieu les principes de fonctionnement des outils de génération de tests focalisés sur la génération de structures de données, avant de voir les outils focalisés sur la génération de séquences de test.

Outils de tests unitaires

Nous décrivons ici la première catégorie d'outils de tests pour Java, basés sur la création de données de test pour le test unitaire de méthodes Java. Différentes techniques sont mises en œuvre, basées sur la résolution de contraintes (TestEra et Korat) ou l'exécution symbolique de code Java (Symstra, et Java PathFinder).

Java PathFinder Les travaux décrits dans [VPK04] s'appuient sur utilisation d'un model-checker symbolique [KPV03] pour la génération de cas de test. Ils utilisent Java PathFinder [HP00], un model-checker explicite, au-dessus duquel est rajoutée une couche permettant l'exécution symbolique.

Cet environnement permet pour générer des structures de données objet Java à partir de la description d'une précondition de méthode, exprimée sous la forme d'une méthode Java retournant une valeur booléenne. Pour ce faire, l'idée est d'appliquer le model-checking sur le code de la méthode pour générer des valeurs permettant d'évaluer la méthode à *vrai*. Cette technique permet ainsi la génération de structures de données non-isomorphes[8], grâce à l'utilisation de solveurs de contraintes ad hoc.

Symstra Symstra [XMSN05] est un outil de génération de tests unitaires pour Java basé sur l'exploration d'états symboliques engendrés par l'exécution symbolique de code Java. Le fonctionnement est basé sur le principe de Java PathFinder, dans lequel des valeurs symboliques sont introduites pour éviter l'énumération exhaustive des valeurs des différents paramètres des méthodes. Symstra propose également des techniques d'élagages de l'exploration de l'espace d'état objet, basées sur la comparaison d'états symboliques. Symstra utilise le prouveur de théorèmes CVCLite [BB04] ou la bibliothèque de résolution de contraintes Omega [KMP+96] pour détecter les inclusions d'espaces d'états symboliques.

Les cas de test produits consistent en une séquence d'invocation de méthodes, de taille définie par l'utilisateur. Symstra réalise l'exploration symbolique du graphe d'états du système et produits des cas de test symboliques constitués d'appels de méthodes avec des paramètres symboliques. Ces paramètres sont ensuite instanciés en utilisant le solveur POOC [SR02], permettant de trouver des valeurs satisfaisant les contraintes relatives aux conditions des branchements empruntés lors de l'exécution symbolique et la création du cas de test.

TestEra TestEra [KM04] est un environnement pour le test de programmes Java, qui automatise à la fois la génération des cas de test et le passage des tests avec l'évaluation de l'oracle. TestEra se focalise sur le calcul de données de test, à la fois pour l'objet hôte sur lequel est invoquée la méthode et sur les paramètres de la méthode. Pour ce faire, TestEra repose sur l'utilisation de l'analyseur Alloy [Jac02]. A partir d'une spécification Alloy des entrées et des invariants du programme Java, Alloy calcule toutes les instances non-isomorphes qui satisfont les spécifications données. De la même manière, une spécification Alloy est utilisée pour décrire la correction des résultats produits par le programme Java.

[8] Deux structures sont isomorphes si elles présentent la même structuration modulo des permutations entre les références d'objets la composant.

TestEra fonctionne à partir d'une spécification TestEra, qui est une combinaison d'un programme Java et de deux spécifications Alloy décrivant respectivement les entrées et les critères de correction pour les résultats obtenus. TestEra nécessite également un mécanisme de concrétisation et d'abstraction respectivement utilisés pour traduire les entrées générées par Alloy en entrées Java, et pour traduire les résultats produits par Java en résultats compréhensibles par Alloy. Le mécanisme est alors le suivant. La spécification Alloy décrivant les entrées est passée à l'analyseur Alloy qui produit des instances satisfaisant les spécifications fournies. Ces instances sont ensuite concrétisées, une par une, pour produire des entrées Java. Ces dernières sont utilisées pour exécuter le cas de test. Les sorties produites par Java sont ensuite abstraites pour se ramener aux formalisme d'Alloy et être confrontées à la spécification des critères de correction des résultats obtenus. Si cette dernière vérification échoue, un contre-exemple est produit. Dans le cas contraire, le processus itère sur l'instance de test suivante.

Les spécifications Alloy décrivant les entrées sont dérivées des préconditions des méthodes et les spécifications décrivant les résultats sont dérivées des postconditions des méthodes.

Korat Korat [BKM02] est le prolongement de TestEra. Réalisé par les mêmes auteurs, il en reprend le principe en se dégageant de la notion d'environnement d'exécution des tests et en l'appliquant des idées de TestEra à Java/JML. Korat propose la génération de cas de test pour des structures de données complexes, mais de taille finie (définie par l'utilisateur). Ces structures satisfont un prédicat Java donné représenté sous la forme d'une méthode Java retournant une valeur booléenne qui décrit les entrées valides. L'exécution de cette méthode est analysée pour indexer les champs qui sont accédés, et dont la valeur influe sur la valeur de vérité du prédicat. Korat s'emploie alors à générer des structures de données candidates, en jouant sur les valeurs des champs analysés. Chacun des candidats est ensuite vérifié par rapport au prédicat de référence. En validant les candidats, les prédicats fournissent des cas de test. La concrétisation des cas de test suppose l'existence de méthodes spécifiques permettant de fixer les valeurs des attributs des objets de manière à construire automatiquement des structures complexes à moindre coût. L'algorithme de génération des valeurs candidates requiert la définition d'une structure nommée "Finitization" qui précise la taille et les domaines des différents attributs des objets sélectionnés. Cette structure permet de réduire l'ensemble des états à un ensemble fini, dans lequel les valeurs candidates vont être sélectionnées. Korat réalise alors la combinaison des valeurs des données pour les champs accédés dans le prédicat. Pour finir, le cas de test est exécuté sur une implantation Java enrichie par les vérifications des assertions JML qui fournissent les résultats attendus.

Korat se démarque de TestEra en implantant son propre mécanisme de génération de données candidates à partir de prédicats Java, nécessitant un effort supplémentaire de la part des utilisateurs pour fournir, en plus du modèle Java/JML, une structure de Finitization permettant de borner la taille des données générées. Pour chaque méthode considérée, le prédicat à satisfaire est donné par la précondition JML traduite en une méthode Java booléenne. La phase de passage de tests est également laissée à Java/JML, qui fournit l'oracle.

Outils de génération de séquences de test

Nous décrivons ici les outils Jartege et Tobias ; ces deux outils sont développés au LSR[9] de Grenoble, dans l'équipe VASCO[10].

Jartege Jartege [Ori05] est un générateur de tests aléatoires pour les programmes Java. Il s'appuie sur une sélection aléatoire des méthodes à tester et une sélection aléatoire des valeurs à fournir en tant que paramètres aux méthodes. Jartege utilise la spécification des méthodes JML, à la fois pour filtrer les exécutions ne répondant pas aux préconditions, et également pour connaître la conformité du programme par rapport aux postconditions des méthodes. La vérification des assertions est ainsi conjointe à la création des séquences de test.

L'approche de Jartege demande à l'utilisateur de fournir la longueur des séquences de test et le nombre de séquences désiré. Il fournit également une fonction de probabilité déterminant la probabilité à laquelle de nouveaux objets doivent être créés pour servir de paramètres aux méthodes exécutées, par rapport au nombre d'objets déjà existant. Cette fonction est par exemple une constante, si l'on souhaite toujours créer un nouvel objet à chaque paramètre, ou une fonction asymptotique qui tend vers 0 d'autant plus rapidement que l'on cherche à minimiser le nombre d'objets présents en mémoire. Intuitivement, cette dernière solution semble la plus favorable pour la détection d'erreurs.

TOBIAS TOBIAS [BMDB+01, LBdB+04] est un outil de génération combinatoire de tests [CDPP96]. Il s'appuie sur un "schéma de test" défini par une expression régulière, décrivant des appels (répétés ou non) à des méthodes ou des groupes de méthodes, à partir d'un contexte initial, et en utilisant des valeurs de paramètres définies par un utilisateur. TOBIAS effectue alors le dépliage du schéma de test et génère toutes les séquences de test correspondantes. TOBIAS permet la génération de séquences de test pour l'outil TGV [JM99], VDM [Jon90] ou Java/JML.

La capacité de TOBIAS à générer un grand nombre de tests à moindre effort, à partir de schémas de test fait sa force, mais également sa faiblesse. En effet, un grand nombre de cas de test peut être généré à partir de schémas de test, mais le risque d'explosion combinatoire lors du dépliage des schémas reste important. La version JML de TOBIAS, présentée dans [LdBMB04], est optimisée par le filtrage des cas de test inconclusifs (i.e., pour lesquels les préconditions des méthodes ne sont pas remplies), dûs à des combinaisons de paramètres non acceptées par les préconditions.

2.3.3 Bilan

Les outils présentés ici pour la génération de tests à partir de JML se décomposent en deux catégories : les outils visant à obtenir un état satisfaisant certaines propriétés relatives à l'objet hôte et aux instances utilisées dans les paramètres. Ces approches sont principalement utilisées pour la création de structures d'objet élaborées, et se destinent

[9] Logiciels Systèmes Réseaux
[10] VAlidation, Spécification et COnstruction de logiciels

majoritairement à être appliquées au test unitaire. A l'inverse, on trouve les méthodes focalisées sur la construction d'une trace d'exécution exerçant au maximum les objets créés par l'invocation de différentes méthodes. Néanmoins, ces approches n'utilisent le modèle JML que pour fournir l'oracle lors de l'exécution.

L'approche présentée dans cette thèse est liée à ces deux catégories, car elle vise à générer des structures objets satisfaisant un prédicat donné extrait du modèle, en invoquant les méthodes des objets créés. Toutefois, notre approche se démarque des autres par l'utilisation complète du modèle JML, à la fois pour définir les objectifs de test, mais aussi pour calculer les séquences d'exécution constituant le cas de test en lui-même.

2.4 Synthèse

Nous avons présenté dans ce chapitre diverses techniques de vérification et de validation de modèles et de programmes, utilisant des approches telles que l'analyse statique, basée sur la preuve, et le test, pour lesquelles différentes déclinaisons ont été présentées. Les approches de test basées sur le modèle permettent de s'abstraire de l'implantation et d'avoir un oracle caractérisant les résultats attendus lors de l'exécution du cas de test sur le système sous test. Les approches basées sur la preuve considèrent JML comme un renfort à la correction du code Java.

L'originalité des travaux présentés dans ce mémoire de thèse découlent de l'application des techniques d'animation à la notation JML. Nous nous appuyons sur un modèle écrit en JML, que nous cherchons à valider, de manière semi-automatique, par animation symbolique à contraintes. De plus, pour que le modèle puisse être utilisé pour la validation de programmes Java à partir d'une approche aux limites, nous avons besoin de garantir un certain nombre de propriétés sur les modèles étudiés. Ces propriétés sont vérifiées en utilisant la programmation logique à contraintes, assimilable à l'analyse symbolique, ou l'animation symbolique du modèle. Une fois le modèle validé pour la génération de test, nous utilisons de la programmation logique avec contraintes. Elle nous sert à déterminer des états contenant des valeurs de test pertinentes, basées sur l'utilisation des valeurs limites. Nous utilisons le mécanisme d'animation symbolique précédemment défini pour générer des séquences de test ayant pour objectif d'atteindre l'état précédemment identifié.

Toutes ces étapes seront définies dans les chapitres relatifs aux contributions, en partie II. Mais avant, nous présentons dans les chapitres suivants les contextes initiaux de cette thèse, à savoir l'environnement d'animation et de génération de tests aux limites BZ-Testing-Tools, et le langage JML ciblé par cette approche.

Chapitre 3

BZ-Testing-Tools

Sommaire

Ce chapitre est consacré à BZ-Testing-Tools [ABC+02, BZT05], abrégé par BZ-TT. Il s'agit d'un environnement permettant l'animation et la génération automatique de tests aux limites à partir de modèles formels B [Abr96] et Z [Spi92]. BZ-TT est le fruit des travaux de recherche de l'équipe Techniques Formelles et à Contraintes du LIFC. BZ-TT repose sur l'utilisation de techniques de résolution et de propagation de contraintes pour réaliser l'évaluation symbolique du modèle formel et le calcul de cas de test fonctionnels à partir de celui-ci.

Ce chapitre débute par la présentation de la méthode BZ-Testing-Tools Nous nous intéressons ensuite au format BZP qui est la notation intermédiaire dans laquelle sont exprimées les notations supportées dans BZ-TT. Nous aborderons ensuite la manière dont les modèles BZP sont interprétés pour réaliser l'animation symbolique. Nous décrirons également dans cette partie la sémantique opérationnelle du format BZP. Nous terminons par un aperçu de la technique de génération de tests qui est une application directe de l'animation symbolique du modèle dans un objectif bien précis : le calcul de séquences de test.

3.1 Présentation générale

L'outil BZ-Testing-Tools [BZT05] repose sur l'évaluation symbolique de modèles formels. Historiquement, les modèles B [Abr96] étaient au cœur de la méthode. A l'heure actuelle, les évolutions successives de la technologie ont permis l'intégration de différentes notations telles que les Statecharts [Har87, HN96] de Statemate [IL005], et UML/OCL [RJB99, WK98].

La méthode BZ-TT a pour finalité la génération automatique de tests aux limites, qui consistent à choisir les valeurs des données de tests aux bornes de leurs domaines. Ces tests sont basés sur un modèle écrit dans une notation formelle. Cette approche est dite "model-based" [Bei95], ce qui signifie que seul le modèle est utilisé pour produire des tests, et il sert d'oracle pour produire le verdict de test. L'architecture de l'outil BZ-Testing-Tools est donné en figure 3.1.

Les notations sources B, Statecharts ou UML, sont traduites dans la *notation intermédiaire*, nommée BZP (B, Z et Prolog), après une analyse syntaxique et un contrôle de types. C'est à partir de ce format que sont réalisées les différentes actions de BZ-TT. L'*animateur*, qui s'appuie sur le *moteur d'exécution* contraint, permet l'exécution et la validation du modèle par le spécifieur. Le *générateur de tests* utilise la résolution de contraintes mise à disposition par les solveurs pour calculer les cibles de test et les données de test. Il produit des séquences de test abstraites en utilisant le moteur d'exécution contraint. Ces tests abstraits sont ensuite automatiquement concrétisés en tests exécutables à partir de *tables de correspondances* (chargées d'établir le lien entre variables du modèle et variables du programme), et de *modèles de scripts de test* permettant de produire différents langages de code. Les tests exécutables sont ensuite passés sur le système sous test dans le but de détecter des non-conformités entre le programme et sa modélisation.

Les travaux ayant donné naissance à BZ-Testing-Tools ont débuté en 1994 au sein du LIFC sous l'impulsion de M. Bruno Legeard [ALL94]. Ils ont été alimentés par de nombreuses thèses. M. Fabrice Ambert [Amb97], M. Laurent Py [Py00], et M. Sébastien Chemin [Che05] ont travaillé à la mise au point de solveurs ensemblistes, coeur de la technologie. M. Fabien Peureux [Peu02] a travaillé à la technique de génération des données de tests. M. Nicolas Vacelet [Vac04] s'est occupé de la définition d'un langage fédérateur, point d'entrée de l'outil, et de l'évaluation symbolique de ce langage par interprétation

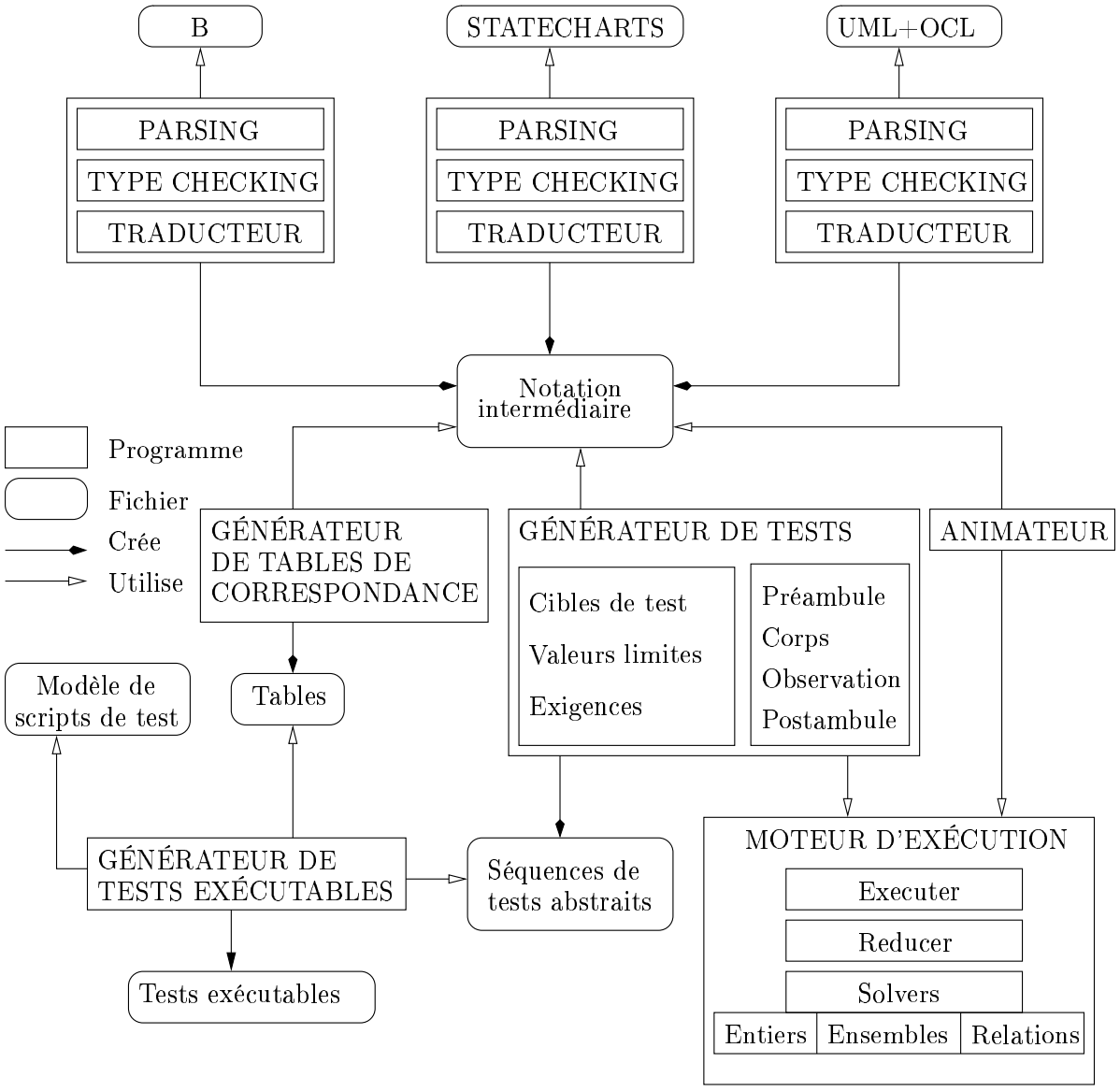


FIG. 3.1 – Architecture complète de l'environnement BZ-TESTING-TOOLS

sous la forme de systèmes de contraintes. Mlle Séverine Colin [Col05] a eu en charge la génération des séquences de tests à l'aide d'algorithmes et d'heuristiques ad hoc pour la recherche d'un état cible dans l'espace d'états. M. Franck Lebeau [Leb05] a travaillé à l'interprétation de la notation Statecharts Statemate comme point d'entrée pour l'outil. Notons également l'habilitation à diriger des recherches soutenue par M. Fabrice Bouquet [Bou05] qui synthétise les travaux sur la sémantique interprétative de modèles formels en animation symbolique et en génération de tests à partir de modèles.

Le format BZP est un format fédérateur pour l'évaluation des notations formelles. C'est dans ce format que sont exprimées les différentes notations prises en compte dans l'environnement BZ-TT. Nous nous intéressons tout d'abord à ce format, et son optimi-

sation, les graphes BGP –Based-Graph Predicates.

3.2 Le format BZP/BGP

Nous présentons dans cette partie la syntaxe du format BZP permettant de déclarer le modèle de données et les transitions. Nous illustrerons ensuite cette description par un exemple, avant de terminer cette partie par la description du format BGP.

3.2.1 Présentation syntaxique

Le format BZP est déclaratif et permet d'exprimer des modèles formels dans une notation pré- et postconditions. Les modèles BZP contiennent des modules, ceux-ci contiennent des données et des opérations. Les données décrivent l'état du système modélisé. Les opérations expriment les transitions et sont écrites avec des prédicats avant/après en logique du premier ordre, où la partie avant du prédicat donne la précondition et la partie après donne la postcondition exprimant les contraintes portant sur l'état d'après du système.

Syntaxe

La déclaration d'un module se fait par l'intermédiaire du prédicat :

$$\texttt{specification(}\mathit{Spec}\texttt{).} \tag{3.1}$$

où *Spec* désigne le nom du module (historiquement nommé "spécification").

Il est à noter que les noms des éléments du BZP (spécification, opérations, variables, constantes) doivent commencer par une minuscule, de manière à se conformer aux conventions des atomes Prolog. De la même manière, chacune des déclarations se termine par un "." en fin de ligne.

Les opérations du modèle BZP sont déclarés par le prédicat

$$\texttt{operation(}\mathit{Spec}, \mathit{Op}\texttt{).} \tag{3.2}$$

où *Spec* représente le nom de la spécification (i.e., du module) et *Op* est le nom de l'opération déclarée.

Les déclarations des données du modèle BZP se font avec le prédicat :

$$\texttt{declaration(}\mathit{Spec}, \mathit{Genre}_D, \mathit{Nom}, \mathit{Type}\texttt{).} \tag{3.3}$$

où *Spec* représente le nom du module, $Genre_D$ est le genre de la donnée choisi dans le tableau 3.1, *Nom* est le nom de la donnée, et *Type* est le type abstrait de la donnée, définis ci-après.

Les prédicats BZP servent à déclarer des propriétés statiques et dynamiques du système. Tout comme pour les machines abstraites B, les invariants sont utilisés à la fois pour définir le typage des données, mais également pour définir les propriétés invariantes du système. Les prédicats du modèle BZP sont exprimés grâce au prédicat :

$$\texttt{predicat(}\mathit{Spec}, \mathit{Genre}_P, \mathit{Id}, \mathit{Pred}\texttt{).} \tag{3.4}$$

Genre de donnée	Sémantique
`static`	constante
`variable`	variable d'état
`input(`*Op*`)`	paramètre d'entrée de l'opération *Op*
`output(`*Op*`)`	paramètre de sortie de l'opération *Op*
`local(`*Op*`)`	variable locale à l'opération *Op*

TAB. 3.1 – Les genres de données du format BZP

où *Spec* représente le module, $Genre_P$ est le genre du prédicat parmi les genres définis dans le tableau 3.2, *Id* est l'identifiant du prédicat, et *Pred* est un prédicat de la logique du premier ordre exprimé à l'aide d'opérateurs logiques, arithmétiques, ensemblistes, fonctionnels et relationnels.

Les prédicats d'initialisation et les postconditions d'opérations sont des prédicats dits avant-après car ils font intervenir les variables d'état avant l'opération et après l'opération. Pour faire la distinction entre les valeurs des variables avant et après l'opération, un opérateur spécifique est introduit, dénoté `prime(x)` (ou `x`′ en notation mathématique) qui permet de référencer la valeur de la variable d'état `x` à l'état après.

Un second opérateur est également spécifique au format BZP. Il est dénoté `choice` (ou [] en notation mathématique) et correspond à un point de choix entre deux prédicats. Cet opérateur sera repris et développé dans la partie suivante.

Genre de prédicat	Sémantique
`static`	assertion
`invariant`	invariant
`initialisation`	initialisation
`pre(`*Op*`)`	précondition de l'opération *Op*
`post(`*Op*`)`	postcondition de l'opération *Op*

TAB. 3.2 – Les genres de prédicats du format BZP

Les références aux variables d'un module se font par l'intermédiaire de l'opérateur `dot`, en employant la syntaxe `Module dot Variable`, dénoté par "." en notation mathématique. Si aucun module n'est précisé, le module par défaut de l'identifiant est le module auquel se rapporte le prédicat considéré (par exemple, dans une précondition, le module contenant l'opération).

Le format BZP autorise également les appels d'opérations. Ces appels sont autorisés dans le cas où l'opération n'a pas d'effets de bord et où elle n'admet qu'une seule valeur de retour. Les appels d'opérations sont réalisés à travers la primitive :

```
call(Spec dot Operation, [ValeurParametre1, ValeurParametre2, ...])
```

Typage

Le format BZP est un format prédicatif destiné à être interprété par Prolog. Il est fortement typé, définissant un typage abstrait et un typage concret des données. Le typage abstrait est défini par la grammaire suivante :

$$T ::= \texttt{atom} \mid \texttt{int} \mid \texttt{set}(T) \mid \texttt{pair}(T, T) \tag{3.5}$$

Le typage concret est lié aux types supportés par les solveurs réalisant l'animation symbolique. Les types des données manipulées sont inspirées du langage B. Le typage concret est défini dans les prédicats BZP sous les traits d'assertions pour les constantes, d'invariants pour les variables d'états, ou de préconditions pour les paramètres d'états.

Exemple 3.1 (Type abstrait et type concrets) Une variable d'état `f` de type abstrait `set(pair(atom,int))` représente une relation, c'est-à-dire un ensemble de couples composés d'un atome et d'un entier. Mais `f` peut plus précisément définir une fonction totale – son type concret – associant chaque atom de l'ensemble {`a`,`b`,`c`} à un entier entre 0 et 10, par l'intermédiaire de la définition de l'invariant adéquat.

```
declaration(s, variable, f, set(pair(atom,int))).
predicat(s, invariant, id1, f ∈ {a,b,c} → 0..10).
```

3.2.2 Un exemple de modélisation BZP

Nous présentons un exemple qui servira d'illustration aux concepts internes à la méthode BZ-TESTING-TOOLS tout au long de ce chapitre. Il s'agit de l'exemple d'un gestionnaire de processus introduit par Dick et Faivre [DF93] pour illustrer la décomposition des modèles formels sous forme de prédicats disjonctifs dont s'inspire la méthode BZ-TT.

La figure 3.2 donne le code BZP de l'exemple. On considère ainsi un gestionnaire de processus qui permet de gérer l'accès à une ressource partagée entre différents processus. Chacun des N processus est désigné par une constante atomique $\texttt{p}_x$ avec $x \in 1..N$. Chacun des processus peut être dans trois états différents : (*i*) en attente lorsqu'il ne demande pas la ressource, (*ii*) prêt lorsqu'il demande l'accès à la ressource mais celle-ci est déjà occupée, (*iii*) actif lorsqu'il accède à la ressource. Nous distinguons ainsi trois variables d'état ensemblistes nommées `waiting`, `ready` et `active` contenant les processus qui sont dans chacun des états respectifs décrits précédemment. L'ensemble des processus est nommé `pID` et sa taille est fixée arbitrairement à 4 pour l'exemple (cf. prédicat 1 Fig. 3.2). L'invariant décrit le fait que chacun de ces ensembles est un sous-ensemble de l'ensemble `pID` (cf. prédicat 2 Fig. 3.2) et que ces ensembles sont disjoints deux à deux (cf. prédicat 3 Fig. 3.2), ce qui représente la propriété d'exclusion mutuelle. Pour finir, le modèle précise qu'au plus un élément est dans l'état actif (cf. prédicat 4 Fig. 3.2) et qu'il n'existe pas de processus qui demande à accéder une ressource alors que celle-ci est libre (cf. prédicat 5 Fig 3.2).

Initialement, aucun processus n'existe, les trois ensembles sont donc vides. La modélisation présente quatre opérations. La première, `new`, permet de créer un processus et de le positionner à l'état d'attente. La deuxième, `del`, est la duale de la précédente, puisqu'elle

```
specification(scheduler).

declaration(scheduler, static, pID, set(atom)).
predicat(scheduler, static, 1, pID = {p1, p2, p3, p4}).

declaration(scheduler, variable, active, set(atom)).
declaration(scheduler, variable, ready, set(atom)).
declaration(scheduler, variable, waiting, set(atom)).

predicat(scheduler, invariant, 2, active ⊆ pID ∧ ready ⊆ pID ∧ waiting ⊆ pID).
predicat(scheduler, invariant, 3, active ∩ ready = ∅ ∧ active ∩ waiting = ∅ ∧
                                  ready ∩ waiting = ∅).
predicat(scheduler, invariant, 4, card(active) ≤ 1).
predicat(scheduler, invariant, 5, active = ∅ ⇒ ready = ∅).

predicat(scheduler, initialisation, 6, active'= ∅ ∧ ready'= ∅ ∧ active'= ∅).

operation(scheduler, new).
declaration(scheduler, input(new), pp, atom).
predicat(scheduler, pre(new), 7, pp ∈ pID).
predicat(scheduler, pre(new), 8, pp ∉ active ∪ ready ∪ waiting).
predicat(scheduler, post(new), 9, waiting' = waiting ∪ {pp}).

operation(scheduler, del).
declaration(scheduler, input(del), pp, atom).
predicat(scheduler, pre(del), 10, pp ∈ waiting).
predicat(scheduler, post(del), 11, waiting' = waiting − {pp}).

operation(scheduler, ready).
declaration(scheduler, input(ready), rr, atom).
predicat(scheduler, pre(ready), 12, rr ∈ waiting).
predicat(scheduler, post(ready), 13, waiting' = waiting − {rr}).
predicat(scheduler, post(ready), 14, (active = ∅ ∧ active' = {rr}) []
                                     (active ≠ ∅ ∧ ready'= ready ∪ {rr})).

operation(scheduler, swap).
predicat(scheduler, pre(swap), 15, active ≠ ∅).
predicat(scheduler, post(swap), 16, waiting' = waiting ∪ active).
predicat(scheduler, post(swap), 17, (ready = ∅ ∧ active'= ∅) []
                                    (∃ pp . pp ∈ ready ∧ active'= {pp} ∧ ready'= ready − {pp})).
```

FIG. 3.2 – Exemple du gestionnaire de processus au format BZP

permet de supprimer un processus qui était en attente. L'opération `ready` permet à un processus en attente de demander l'accès à la ressource. Si aucun processus n'utilise la ressource, alors celle-ci lui est directement assigné ; dans le cas contraire, le processus est considéré dans l'état prêt. Pour finir, l'opération `swap` permet à un processus de libérer la ressource et de revenir à l'état d'attente. S'il existe des processus prêts, alors un de ceux-ci est choisi pour accéder à la ressource. Dans le cas contraire, aucun processus ne passe dans l'état actif.

Le format BZP est purement déclaratif et n'est pas destiné à être écrit par un utilisateur, mais à être généré par un mécanisme de traduction. L'expression des transitions a été optimisée de manière à augmenter la lisibilité et l'interprétation des différents comportements. Nous allons maintenant décrire le format BGP qui nous permettra de décrire l'interprétation par système de contraintes d'un modèle.

3.2.3 Le format BGP

Le format BGP (Based-Graph Predicate) est une représentation des opérations du modèle BZP sous forme arborescente. Ce format est une traduction automatique des modèles BZP.

Le format BGP conserve les informations statiques du BZP, à savoir les déclarations de spécifications, d'opérations, et de données. La différence intervient au niveau des informations dynamiques du BZP : les préconditions et les postconditions des opérations sont conjointes et considérées comme un seul prédicat. En transformant ce prédicat sous la forme normale disjonctive des effets (EDNF) [LPU04], le BGP permet d'exhiber les différents *comportements* d'une opération. La transformation consiste à créer un point de choix entre les prédicats séparés par un opérateur `choice`. Les prédicats issus d'une conjonction sont décrits par un enchaînement d'arcs dans le graphe. Le parcours en profondeur d'un graphe BGP représentant une opération donne les différents comportements composant l'opération considérée.

Ainsi, chaque comportement se décompose en une partie dite "avant" qui ne contient pas de variable primée, et une partie "après", qui contient à la fois des variables primées et des variables non primées.

$$Cpt = Cpt_{avant} \wedge Cpt_{apres} \tag{3.6}$$

Chaque opération est ainsi représentée par un ensemble de comportements, qui est stocké sous la forme d'un graphe. Ce graphe commence au nœud 1 et se termine au nœud 0, comme illustré par les figures 3.3 à 3.6 désignant respectivement les graphes BGP associés aux opérations `new`, `del`, `ready` et `swap` de l'exemple. Pour plus de clarté, les comportements sont également présentés dans le tableau 3.3. Par souci de concision, dans ce tableau, les ensembles *active*, *ready* et *waiting* sont respectivement représentés par leur première lettre a, r et w.

L'animation symbolique des modèles BZP/BGP consiste à interpréter les prédicats rencontrés sur un chemin du graphe entre les nœuds 1 et 0. La partie suivante décrit comment les prédicats sont ainsi interprétés, en mettant en œuvre des techniques de programmation logique avec contraintes.

	Cpt	Partie avant (Cpt_{avant})	Partie après (Cpt_{apres})
new	Cpt_1	$pp \in pID \wedge pp \notin (r \cup w \cup a) \wedge$	$w' = w \cup \{pp\}$
del	Cpt_2	$pp \in w \wedge$	$w' = w - \{pp\}$
ready	Cpt_3	$rr \in w \wedge a = \emptyset \wedge$	$w' = w - \{rr\} \wedge a' = \{rr\}$
	Cpt_4	$rr \in w \wedge a \neq \emptyset \wedge$	$w' = w - \{rr\} \wedge r' = r \cup \{rr\}$
swap	Cpt_5	$a \neq \emptyset \wedge r = \emptyset \wedge$	$w' = w \cup a \wedge a' = \emptyset$
	Cpt_6	$a \neq \emptyset \wedge r \neq \emptyset \wedge pp \in r \wedge$	$w' = w \cup a \wedge a' = \{pp\} \wedge r' = r - \{pp\}$

TAB. 3.3 – Comportements du gestionnaire de processus

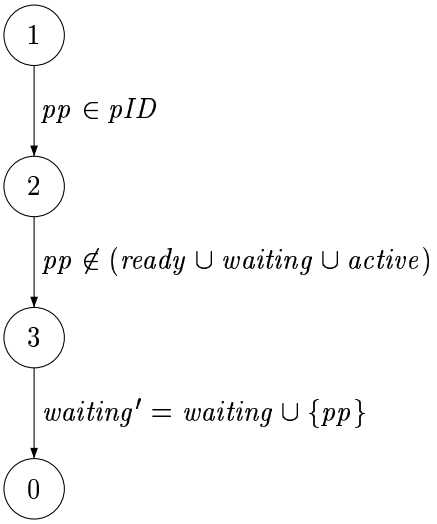


FIG. 3.3 – BGP de l'opération new

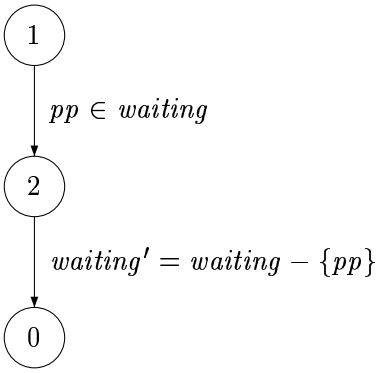


FIG. 3.5 – BGP de l'opération del

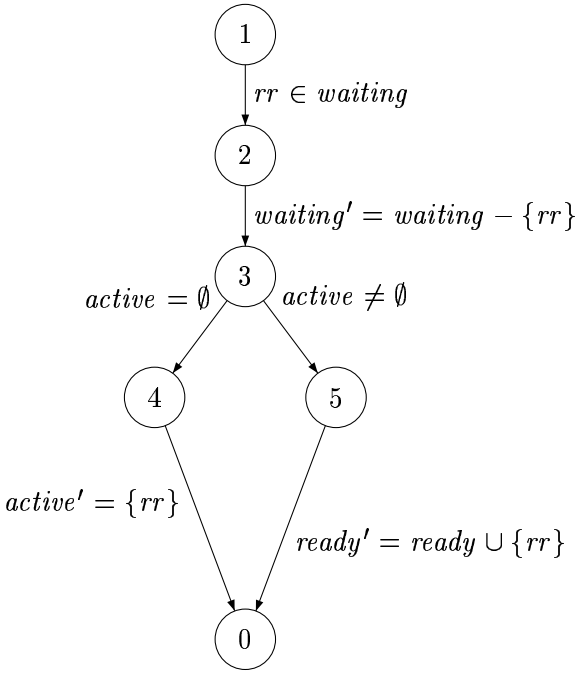


FIG. 3.4 – BGP de l'opération ready

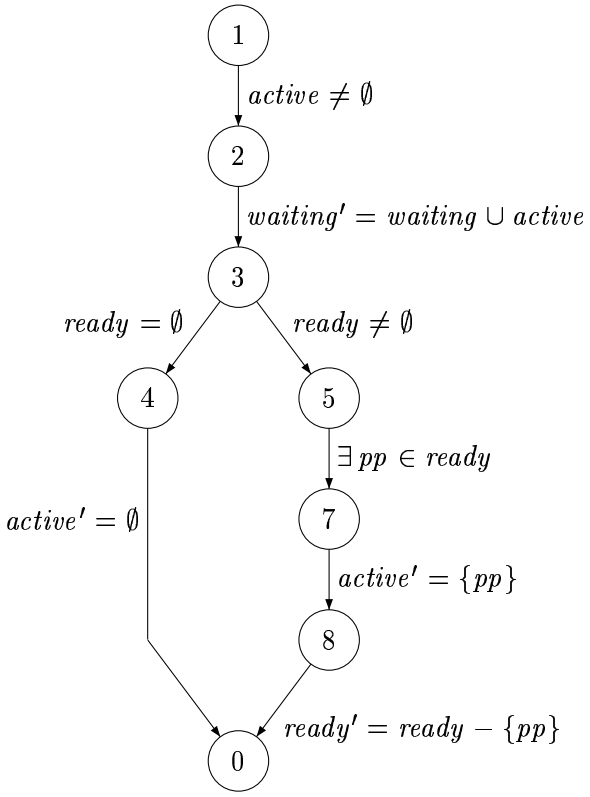


FIG. 3.6 – BGP de l'opération swap

3.3 Sémantique opérationnelle BZP/BGP

La sémantique du BZP/BGP est un système de transition étiqueté (abrégé par la suite par LTS – Labeled Transition System) $\mathcal{L} = \langle \Gamma, A, \longrightarrow \rangle$, où Γ désigne les configurations du système, A l'ensemble des opérations, et $\longrightarrow$ est la relation de transition. Nous allons désormais définir ces notions et leurs applications aux formalismes BZP/BGP.

Une configuration d'exécution BZP/BGP est représentée par un environnement, auquel est associé un système de contraintes. Nous définissons tout d'abord le domaine d'interprétation des variables du store de contraintes.

3.3.1 Domaine d'interprétation

Le domaine d'interprétation des variables manipulées par le solveur CLPS-BZ est l'Univers Homogène Héréditairement Fini, noté $\mathcal{UHHF}$, avec ensembles et couples construit sur les éléments de l'Univers de Herbrand.

Formellement, $\mathcal{UHHF}$ peut être défini de la façon suivante :

$$\mathcal{UHHF} = \bigcup_{n \geq 0}{}_{\bullet} \upsilon^n \tag{3.7}$$

où $\cup_\bullet$ représente l'union disjointe et :

$\upsilon^0 = U_H$ (Univers de Herbrand)

$\upsilon^n = S(\upsilon^{n-1}) \cup_\bullet P(\upsilon^a, \upsilon^b)$ où $a, b \in [0, n-1] \wedge (a = n-1 \vee b = n-1)$

où $S(\upsilon^i)$ représente la collection des ensembles finis composés d'objets appartenant à la collection υ^i. $S(\upsilon^0)$ représente ainsi la collection de tous les ensembles finis construits sur les éléments de l'univers de Herbrand. Similairement, $P(\upsilon^i)$ représente la collection des couples finis composés d'objets appartenant à la collection υ^i.

La propriété d'homogénéité sur les ensembles de la collection $S(\upsilon^i)$ est définie de la manière suivante :

$$\forall\, x \in S(\upsilon^i), \forall\, y, y' \in x . \tau(y) = \tau(y'), i \geq 1 \tag{3.8}$$

où τ est la fonction de type définie sur $\mathcal{UHHF}$ et à valeurs dans l'expression régulière {set,pair}*.{atom}[1] telle que :

$\forall\, x \in \mathcal{UHHF}$

$\tau(x) = atom$ si $x \in \upsilon^0$

$\tau(x) = set(\tau(y))$ si $y \in x \wedge x \in S(\upsilon^i)$

$\tau(x) = pair(\tau(y), \tau(z))$ si $y, z \in x \wedge x \in P(\upsilon^i, \upsilon^j)$

Les objets de la collection υ^0 (l'Univers de Herbrand) sont des éléments atomiques.

3.3.2 Configurations

Nous définissons désormais les configurations du système de transition en commençant par la définition de l'environnement d'exécution.

DÉFINITION 1 (ENVIRONNEMENT D'EXÉCUTION) *Soit $\mathcal{M}$ l'ensemble des modules décrits dans le modèle BZP/BGP, $\mathcal{N}$ l'ensemble des noms de données, $\mathcal{K}$ l'ensemble des*

genres de données définis dans le tableau 3.1, $\mathcal{T}$ l'ensemble des types satisfaisant la grammaire donnée dans l'équation (3.5), et soit $\mathcal{V}$ un ensemble de variables à valeurs dans un ensemble de contraintes $\mathcal{C}_\mathcal{V}$. Un environnement d'exécution Env se définit par

$$Env_\mathcal{V} = \mathcal{M} \times \mathcal{N} \rightarrow (\mathcal{UHHF} \cup \mathcal{V}) \times \mathcal{K} \times \mathcal{T} \tag{3.9}$$

Concrètement, un environnement associe une donnée, identifiée par un module et un nom, à une valeur ainsi qu'un genre et un type. Les valeurs des données sont soit des valeurs de l'ensemble d'interprétation des solveurs, soit une variable qui appartient au store de contraintes.

Nous sommes alors en mesure de définir ce qu'est une configuration de l'ensemble des configurations Γ du système.

DÉFINITION 2 (CONFIGURATION) *Soit $\mathcal{C}_\mathcal{V}$ un store de contraintes mettant en jeu des variables $\mathcal{V}$, et soit $Env_\mathcal{V}$ un environnement d'exécution. Une configuration d'exécution γ d'un modèle BZP/BGP se définit par :*

$$\gamma = \langle \mathcal{C}_\mathcal{V}, Env_\mathcal{V} \rangle \tag{3.10}$$

3.3.3 Transitions et relation de transition

Nous définissons la relation de transition qui permet de passer d'une configuration à une autre lors de l'interprétation du BZP/BGP.

Les transitions du LTS représentant le BZP/BGP sont les opérations déclarées dans le modèle BZP/BGP. Ces opérations possèdent des paramètres d'entrée et de sortie. Chacun de ces paramètres est soit instancié par un utilisateur, soit laissé non renseigné. Dans ce dernier cas, sa valeur sera associée à une variable sur laquelle des contraintes seront posées dans le store de contraintes, notamment des contraintes de typage.

Tous ces concepts sont implantés au sein d'un moteur d'interprétation à contraintes, en charge de gérer les environnements d'exécution, et par extension, les configurations successives lors de l'animation. Nous décrivons à présent ce mécanisme.

3.4 Le moteur d'interprétation à contraintes

Le format BZP/BGP est interprété par une machine virtuelle, implantant la sémantique opérationnelle que nous venons de décrire.

Comme on peut le voir sur la figure 3.1, le moteur d'exécution symbolique est composé de trois modules :

1. Le module **Executer** est le point d'entrée de l'animateur. Il présente l'API d'animation et gère l'environnement d'exécution du modèle.
2. Le module **Reducer** fait le lien entre l'Executer et le Solver. Ce module est en charge de réécrire les prédicats venant de l'Executer dans la syntaxe des solveurs.

3. Le module **Solver** est en charge de la pose des contraintes et de la vérification de la consistance du store de contraintes ainsi obtenu. BZ-TESTING-TOOLS repose sur deux solveurs : le solveur sur les entiers à domaines finis CLP(FD) et CLPS-BZ [BLP02] un solveur dédié à la résolution de contraintes sur les relations et les ensembles homogènes à domaines héréditairement finis.

Le moteur d'animation fonctionne en utilisant la programmation logique avec contraintes. Plus précisément, il utilise les mécanismes de contraintes pour conserver les valeurs des variables, et le mécanisme de "backtrack", issu de la programmation logique, pour effectuer le parcours en profondeur des graphes BGP lors de l'animation.

L'apport de la technologie à contraintes est de réduire considérablement la taille du graphe d'atteignabilité, en regroupant les états valués du système sous la forme d'état symboliques, en fonction de différentes classes d'équivalences. On distingue une classe d'équivalence pour chacun des états permettant des activations similaires du même comportement. A partir d'un état initial déterminé, les états contraints sont engendrés par l'activation d'opérations ayant des paramètres laissés non renseignés. Ceux-ci sont alors contraints par les préconditions –généralement utilisées pour définir le domaine des paramètres– et par les postconditions déterminant un éventuel comportement précis à activer.

Nous décrivons dans cette partie le moteur d'interprétation à contraintes de BZ-TESTING-TOOLS qui implante la sémantique opérationnelle décrite dans la partie précédente. Nous nous intéressons en premier lieu aux éléments de plus bas niveau de l'architecture donnée en figure 3.1. Nous commençons par le module *Solver*, puis nous nous intéresserons aux réécritures réalisées par le module *Reducer*, avant de terminer par les règles d'interprétation écrites dans le module *Executer*.

3.4.1 Le solveur CLPS-BZ

Le solveur CLPS-BZ permet l'acquisition et l'évaluation de contraintes qui sont initialement caractérisées par un prédicat logico-ensembliste (notamment basées sur des structures d'ensembles et de couples). Ainsi, un prédicat $P(v_1, v_2, \ldots, v_n)$ s'applique aux variables $v_1, v_2, \ldots, v_n$, ayant comme ensembles de définition respectifs $dom(v_1)$, $dom(v_2)$, ..., $dom(v_n)$. Calculer la contrainte formulée par le prédicat P revient à calculer l'ensemble des n-uplets $\langle x_1, x_2, \ldots, x_n \rangle$ où chaque x_i est un élément de l'ensemble de valeurs $dom(v_i)$, et où $P(x_1, x_2, \ldots, x_n)$ est vérifié. Un tel n-uplet est une solution du système de contraintes. Le traitement des entiers est assuré par le solveur numérique CLP(FD).

Le solveur CLPS-BZ [Che05] utilise une consistance d'arc AC-3+ [Apt03, JM94], de complexité $\mathcal{O}(ek^3)$, où e est le nombre de contraintes et k le cardinal du plus grand domaine. L'obtention des solutions d'un système de contraintes utilise un algorithme de forward-checking de complexité $\mathcal{O}(ek^2)$. Le solveur CLPS-BZ n'assure qu'une consistance d'arcs du système de contraintes. De ce fait, la cohérence du graphe de contraintes (où les sommets et les arcs sont respectivement les variables et les contraintes) n'est que partielle : la cohérence des paires de variables mise en relation par au moins une contrainte est vérifiée, mais la cohérence de trois variables reliées 2 à 2 ne l'est pas. Dès lors, le

graphe de contraintes peut être inconsistant sans nécessairement être détecté comme tel. La complétude est garantie par la recherche d'un n-uplet solution du graphe de contraintes portant sur l'ensemble des variables. Cette technique est coûteuse en temps.

Les solveurs gèrent un système de contraintes, acquises sous la forme de prédicats écrits dans leur syntaxe. La syntaxe de CLP(FD) couvre les opérateurs arithmétiques du BZP, à savoir l'addition, la soustraction, la multiplication, la division entière et le reste, ainsi que les opérateurs de comparaison supérieur (ou égal), inférieur (ou égal) et (in)égalité. Les opérateurs du solveur CLPS-BZ ainsi que leur définition logico-ensemblistes sont donnés dans le tableau 3.4.

Primitive	Définition	Définition
A eq B	Egalité	$A = B$
A neq B	Inégalité	$A \neq B$
A ins S	Appartenance	$A \in S$
A nin S	Non appartenance	$A \notin S$
S sub T	Inclusion	$S \subseteq T$
S # N	Cardinalité	$N = card(S)$
rdom(Q,S,R)	Restriction de domaine	$Q = S \lhd R$
dom(S,R)	Domaine	$S = dom(R)$
ran(S,R)	Co-domaine	$S = ran(R)$
inv(Q,R)	Relation inverse	$Q = {}^{\sim}(R)$
power(T,S)	Ensemble des parties	$T = \mathbb{P}(S)$
pcart(S,T,U)	Produit cartésien	$S = T \times U$
couple(X,Y)	Couple	$X \mapsto Y$
S union T	Union	$S \cup T$
S inter T	Intersection	$S \cap T$
S setminus T	Différence ensembliste	$S \setminus T$

TAB. 3.4 – Opérateurs du solveur CLPS-BZ

Pour garantir la consistance du store, le solveur CLPS-BZ dispose d'une primitive `labeling` qui permet de résoudre le système de contraintes pour tenter de trouver une instanciation des variables contenues dans le store.

3.4.2 Le module Reducer

Les règles de réécriture des opérateurs BZP non supportés par le solveur sont données par le tableau 3.5. Les différents opérateurs dans lesquels sont réécrits les opérateurs BZP sont donnés dans le tableau 3.4.

Le module Reducer implante ces règles de réécritures. Il est en charge de les appliquer sur les prédicats qu'il reçoit du module Executer et d'envoyer les prédicats réécrits au Solver. Durant la réécriture, ce module réalise également le remplacement des expressions désignant des variables d'états par la valeur ou la variable CLPS associée, extraite de l'environnement d'exécution courant.

Opérateur	Définition	Réécriture
$P_1 \Rightarrow P_2$	Implication	$\neg P_1 \lor P_2$
$P_1 \Leftrightarrow P_2$	Equivalence	$(\neg P_1 \land \neg P_2) \lor (P_1 \land P_2)$
$S \not\subseteq T$	Non inclusion	$\exists X / X \in S \land X \notin T$
$S \subset T$	Inclusion stricte	$S \subseteq T \land S \neq T$
$S \not\subset T$	Non inclusion stricte	$S \subseteq T \land card(S) < card(T)$
$S \rhd T$	Restriction de co-domaine	$(T \lhd S^{-1})^{-1}$
$S \lhd T$	Soustraction de domaine	$(dom(T) \setminus S) \lhd T$
$S \rhd T$	Soustraction de co-domaine	$S \rhd (ran(S) \setminus T)$
$S \lhd\!\!+ T$	Surcharge gauche	$(dom(T) \lhd S) \cup T$
$S[T]$	Image relationnelle	$ran(T \lhd S)$
$\mathbb{P}1(S)$	Ensemble des parties non vide	$\mathbb{P}(S) \setminus \emptyset$

TAB. 3.5 – Réécriture des opérateurs BZP

3.4.3 Le module Executer

Le module Executer est le cœur du mécanisme d'animation contraint puisqu'il est en charge de la gestion des environnements d'exécution successifs et du parcours des graphes BGP pour permettre le calcul des états successifs lors de l'animation.

L'Executer réalise l'initialisation de l'environnement d'exécution en créant les entrées correspondants aux différentes variables d'états décrites dans le fichier BZP. Il utilise ensuite les graphes BGP calculés à partir des prédicats (clause `predicat/4`) issus du BZP pour faire évoluer ces environnements. Les graphes BGP sont parcourus en profondeur, et chaque prédicat rencontré est envoyé au module Reducer. Si l'ajout du prédicat au système de contraintes existant provoque une inconsistance, un mécanisme de "backtrack" se déclenche pour remonter au dernier point de choix dans le parcours du graphe et sélectionner le branchement suivant.

Ce mécanisme est implantée dans une primitive nommée *execute_graph*, et appliquée successivement aux prédicats :

- `static` et `initialisation` : pour calculer la configuration initiale.
- `pre` et `post` : pour calculer la configuration résultante de l'exécution d'une transition. Cette étape est précédée de l'ajout dans l'environnement des entrées correspondant aux variables locales à l'opération exécutée, étape elle-même précédée du retrait des variables locales introduites pour l'exécution de l'opération précédente.

La primitive *execute_graph* est également appelée lors de l'interprétation par le module Reducer d'un appel dynamique à une opération, par l'instruction BZP `call`. Dans ce cas, le traitement effectué sur l'environnement d'exécution est le suivant. (*i*) sauvegarde des paramètres d'entrée de l'environnement courant, (*ii*) création des paramètres d'entrée de l'opération appelée, (*iii*) exécution du graphe correspondant à l'opération appelée, (*iv*) récupération de la valeur du paramètre de sortie, et (*v*) restauration des entrées sauvegardées précédemment.

Dans le cas où aucune branche du graphe BGP n'a pu être exécutée, l'exécution de

l'opération échoue et l'état suivant n'existe donc pas. Illustrons ces propos avec l'animation de l'exemple du gestionnaire de processus introduit précédemment.

Animation symbolique de l'exemple

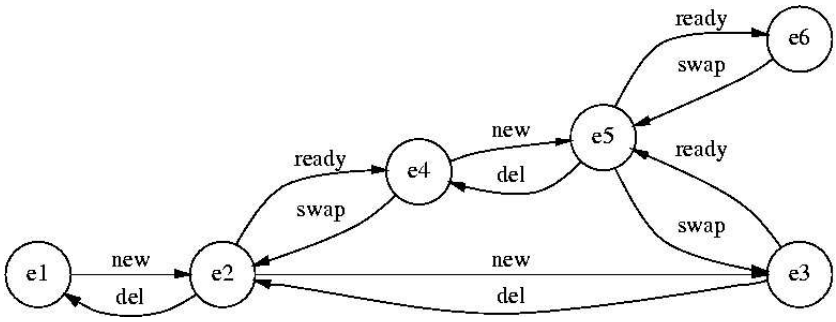


FIG. 3.7 – Animation symbolique de l'exemple avec 2 processus

Le graphe d'atteignabilité de l'animation symbolique de l'exemple est donné en figure 3.7. En guise de comparaison, et pour mesurer les apports de l'animation symbolique, la figure 3.8 présente le graphe d'atteignabilité issu de l'animation valuée du même exemple.

Alors que le graphe contraint ne représente que 6 configurations symboliques, le graphe valué contient 10 configurations, pour un ensemble de processus `pID` de cardinalité 2. Pour 4 processus, comme décrit dans l'exemple présenté en figure 3.2, nous obtenons 15 configurations contraintes pour 124 configurations valuées. Le tableau 3.6 expose la réduction du nombre d'état en fonction de la taille de l'ensemble des processus PID.

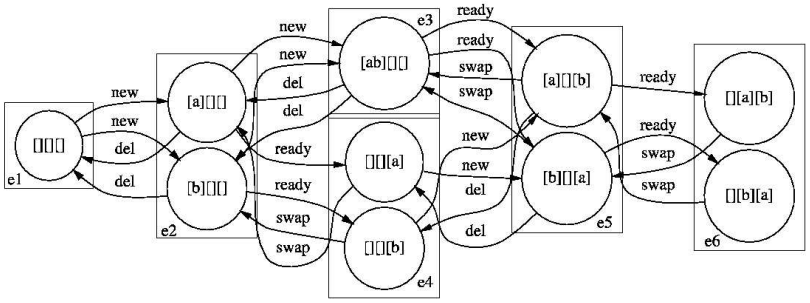


FIG. 3.8 – Animation valuée de l'exemple avec 2 processus

Optimisations des graphes BGP

Les travaux de thèse de M. Nicolas Vacelet [Vac04] ont, entre autres, permis de définir des optimisations concernant l'interprétation des prédicats contenus dans le graphe. Cette nécessité est apparue avec la volonté de prendre en compte de modèles industriels.

Les optimisations majeures sont au nombre de 4 :

1. Factorisation des sous-expressions communes : les sous-expressions qui sont répétées dans un graphe sont évaluées une seule fois en début d'exécution du graphe et leur résultat est stocké dans une variable locale, qui est rappelée ensuite.

$card(PID)$	**nombre d'états contraints**	**nombre d'états valués**
1	3	3
2	6	10
3	10	35
4	15	124
5	21	437
6	28	1522
7	36	5231
n	$\frac{n^2+3n+2}{2}$	$> 3^n$

TAB. 3.6 – Etude comparative du nombre d'états contraints/valués

2. Elimination de comportements inconsistants : les comportements du graphe qui sont inconsistants, c'est-à-dire contenant des prédicats contradictoires entre eux, sont éliminés du graphe BGP.
3. Réécriture des $A \vee B$: à l'exécution, les disjonctions sont réécrites sous la forme $A \wedge \neg B$, $\neg A \wedge B$, $A \wedge B$ qui représentera le moins de points de choix par la suite.
4. Retardement des points de choix : les points de choix sont retardés le plus possible dans le graphe BGP. Ainsi le graphe BGP est ré-ordonné et réécrit avec le nouvel ordonnancement retardant les points de choix dans le graphe.

3.5 La génération de tests aux limites

Cette partie décrit l'application de la résolution de contraintes et de l'évaluation symbolique dans l'objectif de générer des cas de tests fonctionnels à partir de modèles traduits dans le format BZP/BGP. Nous décrivons dans cette partie la méthode BZ-TESTING-TOOLS.

3.5.1 La méthode BZ-TESTING-TOOLS

Le principe de la méthode BZ-TESTING-TOOLS est représenté sur la figure 3.9. Cette méthode a pour objectif la génération automatique de cas de tests fonctionnels, i.e., boite noire.

Un modèle est produit à partir d'un cahier des charges ou des spécifications informelles des besoins. Parallèlement, le programme est développé à partir du même document. Si ces deux processus sont indépendants, la méthode BZ-TT requiert néanmoins que le programme et la modélisation partagent des "points d'entrée" sous la forme d'opérations ayant des signatures proches pour que l'exécution des tests soit simple. De plus, le programme doit fournir des opérations dites "d'observation" qui vont permettre de confronter les valeurs attendues par le modèle et celles du système sous test. Le processus de génération des tests est le suivant.

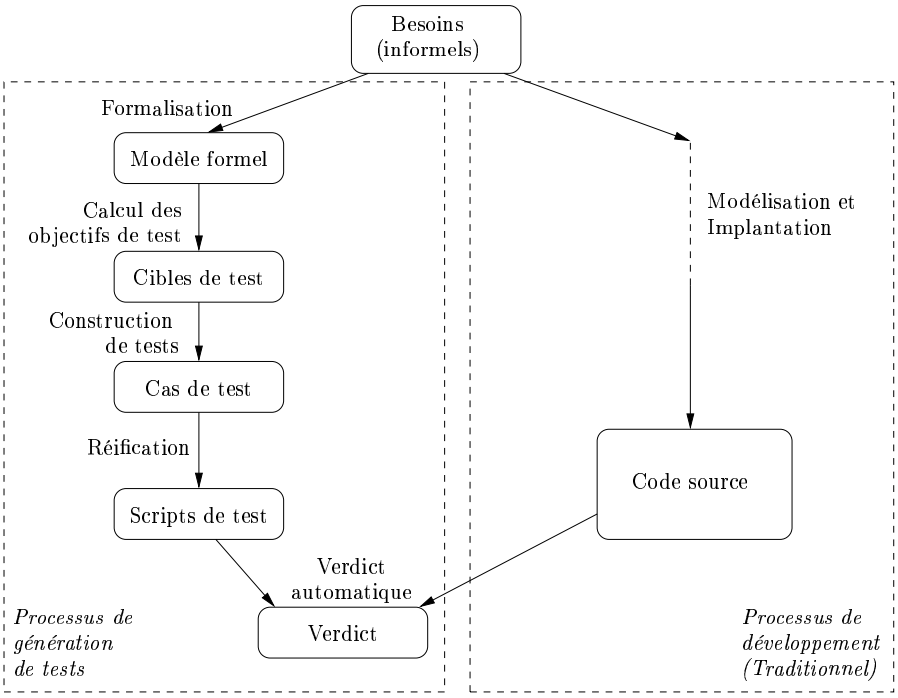


FIG. 3.9 – Démarche de génération de tests de la méthode BZ-TESTING-TOOLS

A partir du modèle, les *cibles de tests* sont calculées. Ils expriment l'activation d'un comportement du système dans un état pertinent. Cet état pertinent est un état aux limites, c'est-à-dire qu'au moins une des variables d'état du système est à un extremum (minimum ou maximum) de son domaine. Les cibles de test ainsi produites contiennent une description de l'état du système pertinent, et les valeurs des paramètres pour l'appel à l'opération testée. La *construction des tests* vise ainsi à calculer une séquence d'opérations depuis l'état initial, qui va permettre de placer le système dans l'état pertinent. Cette phase est nommée calcul de préambule. L'opération sous test est ensuite exécutée. Pour permettre de produire le verdict du test, les opérations d'observations du système sont invoquées et les résultats obtenus seront comparés avec l'oracle. Pour finir, le cas de test se compose d'un postambule qui permet d'enchaîner les cas de tests, soit en réinitialisant le système, soit en le plaçant dans un état permettant l'activation d'une autre opération. Cette phase produit des cas de tests abstraits qui sont ensuite *réifiés* [BL03] pour produire du code exécutable dans le format des bancs sous test. Lors de l'exécution, les appels aux opérations d'observation donneront les valeurs des variables d'états du système au moment de l'exécution. Celles-ci pourront alors être comparées aux valeurs calculées lors de la construction du cas de test. Ce processus est détaillé dans le reste de cette partie.

3.5.2 Calcul des cibles de test

Le but des tests à partir des modèles BZP/BGP vise à activer les comportements de chacune des opérations du modèle. Chacun de ces comportements est issu des graphes BGP représentant chacune des opérations du modèle. L'extraction des cibles de test est le fruit du travail de thèse de M. Fabien Peureux [Peu02].

Couverture des décisions

Dans le but de satisfaire à des critères de couverture du modèle, les disjonctions présentes dans les décisions du modèle formel sont réécrites. Par décision, nous entendons les conditions de branchement dans le graphe BGP des opérations. Lorsque les critères de couverture sont remplis, les cas de test produits garantissent la couverture du modèle. On distingue 4 réécritures, chacune permettant de satisfaire un critère de couverture différent, comme illustré par la figure 3.10.

Réécriture 1 La première réécriture consiste à laisser les disjonctions telles qu'elles.

$$P_1 \lor P_2 \rightsquigarrow P_1 \lor P_2 \tag{3.11}$$

Cette réécriture permet de satisfaire le critère *statement coverage* qui correspond à la couverture de tous les nœuds du graphe de flot de contrôle. Elle répond aussi au critère *decision coverage*, qui correspond à la couverture de tous les branchements conditionnels d'un graphe de flot de contrôle.

Réécriture 2 La deuxième réécriture consiste à considérer chacun des prédicats disjonctifs de manière indépendante, sans garantir que ces deux branches soient mutuellement exclusives.

$$P_1 \lor P_2 \rightsquigarrow P_1 \,[]\, P_2 \tag{3.12}$$

Cette réécriture permet de satisfaire les critères précédents, mais elle satisfait en plus le critère *condition coverage* puisque, par définition, toutes les valeurs d'une condition sont évaluées à vrai et à faux.

Réécriture 3 La troisième réécriture consiste à considérer chacun des prédicats disjonctifs de manière exclusive garantissant que ces deux branches soient mutuellement exclusives.

$$P_1 \lor P_2 \rightsquigarrow P_1 \land \neg P_2 \,[]\, \neg P_1 \land P_2 \tag{3.13}$$

Cette réécriture permet de satisfaire les critères précédents, mais elle satisfait en plus le critère *full predicate coverage* puisque, par définition, chaque décision et chaque condition

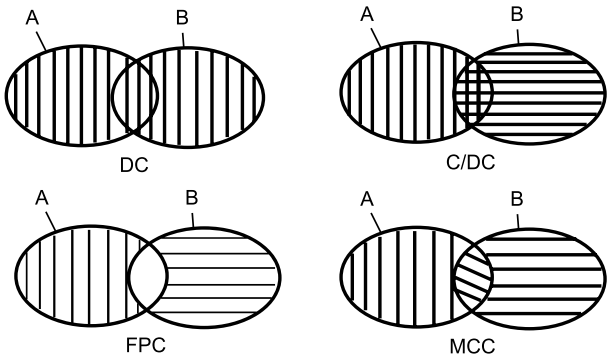


FIG. 3.10 – Illustration des critères de couvertures relatifs aux réécritures

dans cette décision a pris chaque valeur au moins une fois et chaque condition a affecté sa décision [OXL99].

Réécriture 4 La quatrième réécriture consiste à considérer toutes les valeurs de vérité possibles pour une disjonction.

$$P_1 \lor P_2 \rightsquigarrow P_1 \land P_2 \,[]\, P_1 \land \neg P_2 \,[]\, \neg P_1 \land P_2 \tag{3.14}$$

Cette réécriture permet de satisfaire les critères précédents, mais elle satisfait en plus le critère *multiple condition coverage* puisque, par définition, pour chaque condition, chaque valeur est explorée.

L'application de la réécriture à un graphe BGP produit un nouveau graphe BGP, résultat de l'application du dépliage des disjonctions souhaité.

Prédicats de spécialisation

Les cibles de test donnent la possibilité à un utilisateur de définir les états qu'il juge pertinents pour chacun des tests sous la forme d'un prédicat dans la syntaxe BZP, appelé *prédicat de spécialisation* et dénoté P_{spe}.

Par exemple, l'utilisateur peut vouloir tester tous les comportements lorsqu'il y a au moins 2 processus en attente. Il obtiendra ainsi les cibles de test du tableau 3.7. Le solveur CLPS-BZ permet de détecter et d'éliminer les cibles de test inconsistantes.

Opération	Cpt_{avant}	P_{spe}
new	$pp \in PID \land pp \notin (ready \cup waiting \cup active) \land$	$card(waiting) > 2$
del	$pp \in waiting \land$	$card(waiting) > 2$
ready	$rr \in waiting \land active = \{\} \land$	$card(waiting) > 2$
	$rr \in waiting \land active \neq \{\} \land$	$card(waiting) > 2$
swap	$active \neq \{\} \land ready = \{\} \land$	$card(waiting) > 2$
	$active \neq \{\} \land ready \neq \{\} \land pp \in ready \land$	$card(waiting) > 2$

TAB. 3.7 – Cibles de test du gestionnaire de processus avec $P_{spe} = card(waiting) > 2$

Le prédicat de spécialisation P_{spe} peut être calculé aux limites du comportement, comme nous allons le voir maintenant.

Calcul des cibles de test

La méthode BZ-TT permet de considérer des états limites pour l'activation des comportements. Un état aux limites se définit comme un état pour lequel une des variables locales est à un extremum (soit au minimum, soit au maximum) de son domaine.

Pour chacun des types est associé une fonction d'optimisation utilisée pour calculer les buts aux limites. Cette fonction permet de définir sur quel composante s'applique l'idée de limites, et ce, pour les différents types de données possibles dans le format BZP.

Les buts aux limites sont calculés en minimisant et/ou maximisant, sur les contraintes de l'invariant, une fonction d'optimisation $f(V_i) = \sum_{v \in V_i} card(v)$ où V_i définit un vecteur

de variables d'état (une quantification existentielle est posée sur les éventuelles variables d'entrée des opérations). Les valeurs ainsi calculées enrichissent la cible de test en étant intégrées dans le prédicat de spécialisation associé au comportement testé.

Les résultats de ce calcul sur l'exemple du gestionnaire de processus, décrit en figure 3.2 sont présentés dans le tableau 3.8. Ce tableau reprend chacun des comportements décrits dans le tableau 3.3, chacune des cibles de tests est ainsi composée de la partie d'activation du comportement et du prédicat de spécialisation définissant les valeurs limites précédemment calculées.

Test	Cpt_{avant}	P_{spe}
Ct_1	$Cpt^1_{avant} \wedge$	$card(waiting) = 0 \wedge card(ready) = 0 \wedge card(active) = 0$
Ct_2	$Cpt^1_{avant} \wedge$	$card(waiting) = 3 \wedge card(ready) = 0 \wedge card(active) = 0$
Ct_3	$Cpt^1_{avant} \wedge$	$card(waiting) = 0 \wedge card(ready) = 2 \wedge card(active) = 1$
Ct_4	$Cpt^1_{avant} \wedge$	$card(waiting) = 2 \wedge card(ready) = 0 \wedge card(active) = 1$
Ct_5	$Cpt^1_{avant} \wedge$	$card(waiting) = 1 \wedge card(ready) = 1 \wedge card(active) = 1$
Ct_6	$Cpt^2_{avant} \wedge$	$card(waiting) = 1$
Ct_7	$Cpt^2_{avant} \wedge$	$card(waiting) = 4$
Ct_8	$Cpt^3_{avant} \wedge$	$card(waiting) = 1 \wedge card(active) = 0$
Ct_9	$Cpt^3_{avant} \wedge$	$card(waiting) = 4 \wedge card(active) = 0$
Ct_{10}	$Cpt^4_{avant} \wedge$	$card(waiting) = 1 \wedge card(active) = 1$
Ct_{11}	$Cpt^4_{avant} \wedge$	$card(waiting) = 3 \wedge card(active) = 1$
Ct_{12}	$Cpt^5_{avant} \wedge$	$card(ready) = 0 \wedge card(active) = 1$
Ct_{13}	$Cpt^5_{avant} \wedge$	$card(ready) = 0 \wedge card(active) = 1$
Ct_{14}	$Cpt^6_{avant} \wedge$	$card(ready) = 1 \wedge card(active) = 1$
Ct_{15}	$Cpt^6_{avant} \wedge$	$card(ready) = 3 \wedge card(active) = 1$

TAB. 3.8 – Cibles de test du gestionnaire de processus avec les buts aux limites

Il convient de noter que d'autres fonctions d'optimisation peuvent être employées. Par exemple, à partir du Cpt^1_{avant} en utilisant la fonction d'optimisation $f(V_i) = \sum_{v \in V_i} card(v^2)$ sur les variables d'états V_i en maximisation, seule la cible de test Ct_2 est produite, et avec la fonction d'optimisation $f(V_i) = \sum_{v \in V_i} \sqrt{card(v)}$, seule la cible Ct_5 est produite (voir [Peu02]).

3.5.3 Construction des cas de test

Un cas de test est divisé en quatre sous-traces composées d'activation d'opérations que présente la figure 3.11 :

Preamble : cette sous-trace permet d'atteindre une cible de test depuis l'état initial du système.

Body : cette sous-trace consiste à activer le comportement lié à la cible de test avec des valeurs extrema d'entrée.

Observation : cette sous-trace est constituée par l'activation d'opérations d'observation précisant l'état attendu du système pour l'assignation d'un verdict (pass/fail).

Postamble : cette sous-trace permet depuis l'état courant d'atteindre l'état initial du système. Ceci permet d'enchaîner le passage des cas de test.

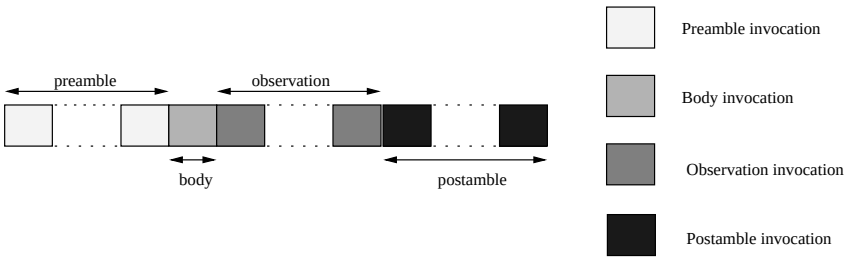


FIG. 3.11 – Constitution d'un cas de test

Par exemple, pour l'exemple du gestionnaire de processus, 6 tests, présentés dans le tableau 3.9, peuvent être construits à partir des cibles de test les plus simples, c'est-à-dire à partir de la partie avant de ses comportements.

Tests	Préambule	Body	Postambule
t_1		$new(p1)$	$del(p1)$
t_2	$new(p1)$	$del(p1)$	
t_3	$new(p1)$	$ready(p1)$	$swap$, $del(p1)$
t_4	$new(p1)$, $new(p2)$, $ready(p1)$	$ready(p2)$	$swap$, $swap$, $del(p1)$, $del(p2)$
t_5	$new(p1)$, $ready(p1)$	$swap$	$del(p1)$
t_6	$new(p1)$, $new(p2)$, $ready(p1)$, $ready(p2)$	$swap$	$swap$, $del(p1)$, $del(p2)$

TAB. 3.9 – Tests basiques générés pour le gestionnaire de processus

Les séquences de tests abstraites sont calculées en utilisant le mécanisme d'animation symbolique, mettant en application des algorithmes de recherche de différents types en fonction de l'objectif à atteindre. Ces travaux ont fait l'objet de la thèse de Mlle Séverine Colin [Col05]. Ces travaux ont étudié l'application de différents algorithmes pour permettre la construction des cas de tests. On distingue deux types d'algorithmes ayant des visées différentes :

1. Les algorithmes de type "meilleur d'abord" (Best-First). Ces algorithmes ont pour objectif de rechercher un état particulier satisfaisant un ensemble de contraintes (définissant un but aux limites). C'est ce type d'algorithme qui est employé pour calculer un préambule menant à un état aux limites. Ces algorithmes sont de complexité $\mathcal{O}(k^n)$ où k désigne le nombre de comportements et n désigne la profondeur à laquelle on borne la recherche. Ils sont donc assez coûteux et leur utilisation est à réserver pour chercher à atteindre des cibles bien particulières.

2. Les algorithmes de type "parcours en largeur sans remise" qui consistent à couvrir l'activation du plus grand nombre de comportements. Ces algorithmes s'utilisent principalement lorsque l'objectif de test ne considère pas de prédicat de spécialisation $P_{spe} = true$, car ils révèlent surtout leur efficacité sur la couverture des comportements, par opposition aux états cibles, à l'instar de la stratégie précédente.

3.6 Synthèse

Nous avons présenté dans ce chapitre l'approche BZ-Testing-Tools, qui est à la fois une méthode visant à générer des tests aux limites et un outil implantant un animateur de modèles formels et un générateur de tests aux limites. Nous avons présenté son architecture, basée sur un format intermédiaire, fédérateur de plusieurs notations, et possédant une sémantique de système de transitions exprimé sous la forme pré- et postconditions.

Nous avons ensuite décrit le fonctionnement de l'animation symbolique des modèles BZP, basée sur la transformation en graphe BGP des prédicats avant/après des opérations du modèle BZP. Ceci nous a permis d'aborder la technique de génération de test dits "aux limites" de BZ-Testing-Tools. Cette technologie a été utilisée dans de nombreux projets industriels, notamment dans le cadre de la validation d'applications sur carte à puce [LP01, BLL04, BLLP04]. Forte de cette expérience, la technologie BZ-Testing-Tools a fait l'objet d'un transfert de technologies dans le cadre de la loi sur l'innovation de 1999 et a ainsi conduit à la création de l'entreprise LEIRIOS Technologies [LEI05] en 2003.

Le projet BZ-Testing-Tools représente le contexte scientifique initial de cette thèse. Le chapitre suivant s'intéressera à la notation Java Modeling Language qui est le langage de modélisation objet que nous avons étudié durant cette thèse.

Chapitre 4

Le Java Modeling Language

Sommaire

Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie).

Dave Small, Extrait de ST Magazine

Le Java Modeling Language [LBR98, LBR99, LBR02], abrégé par JML, est le langage de modélisation formelle associé au langage de programmation Java. JML est né des réflexions de Gary T. Leavens au milieu des années 90. Il s'agit d'un langage d'assertions qui s'inspire d'Eiffel [Mey97], dont il se veut le descendant direct. Les annotations JML sont ainsi écrites au niveau du code Java, soit dans une classe, soit dans une interface. Ces annotations permettent de décrire un modèle statique en exprimant des invariants, et un modèle dynamique en décrivant les comportements des méthodes considérées à l'aide de préconditions et de postconditions.

Le principe essentiel de JML est désigné sous le terme "conception par contrat". Ceci signifie que le système doit remplir les préconditions d'une méthode lors de l'invocation de celle-ci. En contrepartie, le système s'engage à remplir les postconditions de la-dite méthode. Cette entente est donc contractuelle : le système s'engage à remplir un contrat lors de l'appel à la méthode, et la méthode s'engage à établir la postcondition.

La syntaxe de JML est basée sur la syntaxe de Java. Les annotations JML sont embarquées au sein des classes Java qu'elles décrivent, dans des blocs de commentaires spécifiques. Les blocs d'annotations JML sont soit sur une ligne commençant soit par `//@`, soit ils sont délimités par `/*@` et `@*/`. C'est à l'intérieur de ces blocs JML que le modèle en lui-même est exprimé.

Les classes annotées en JML sont nommées des *spécifications de type*. Cette appellation est cohérente puisque les classes Java représentent des types de données réutilisables. Ces spécifications de type décrivent les propriétés statiques du système. Elles sont complétées par des *spécifications de méthodes* qui décrivent le comportement dynamique du modèle, en explicitant le comportement des méthodes.

Ce chapitre s'intéresse à introduire le langage JML. Nous allons commencer par une présentation syntaxique de celui-ci, introduisant des notions qui seront par la suite réutilisées ou détaillées. Nous aborderons ensuite les spécifications de type, puis nous nous attarderons sur la spécification des méthodes. Nous présenterons ensuite un exemple de modèle écrit avec JML qui nous servira à illustrer les contributions de cette thèse. Pour finir, nous verrons deux outils dédiés à JML, le Runtime Assertion Checker et JMLUnit.

4.1 Quelques notions de JML

L'objectif de cette partie n'est pas de donner une description complète des modèles JML, mais plutôt de donner un aperçu des possibilités du langage.

Cette partie décrit les principes de spécification en JML, c'est-à-dire la description des possibilités de modélisation d'une classe. Nous commencerons par voir quelques notions relatives à JML, telles que la syntaxe des expressions JML. Nous présenterons ensuite le fonctionnement de la modélisation JML, i.e., l'utilisation de *clauses*.

4.1.1 Expression des prédicats JML

Les prédicats JML sont exprimés à partir de la syntaxe Java, étendue d'un certain nombre d'opérateurs spécifiques. Parmi les plus usités, nous retiendrons les opérateurs

des trois catégories suivantes : les opérateurs booléens, les opérateurs de postcondition et les opérateurs relatifs aux objets.

Les opérateurs booléens

Les opérateurs booléens sont listés dans le tableau 4.1. Il s'agit des symboles d'implication, d'équivalence et des quantificateurs.

Opérateur	Nom	Définition
`==>`	Implication	$\Rightarrow$
`<==>`	Équivalence	$\Leftrightarrow$
`(\forall` T_1 v_1`;` $P_1(v_1)$ `;` $P_2(v_1)$`)`	Quant. universelle	$\forall\, v_1.(v_1 \in T_1 \wedge P_1 \Rightarrow P_2)$
`(\exists` T_1 v_1`;` $P_1(v_1)$ `;` $P_2(v_1)$`)`	Quant. existentielle	$\exists\, v_1.(v_1 \in T_1 \wedge P_1 \wedge P_2)$

TAB. 4.1 – Tableau des opérateurs booléens de JML

Les opérateurs de postconditions JML

Les mots-clés JML sont distingués des mots-clés Java usuels par un préfixe \. Ceux-ci augmentent le pouvoir d'expressivité des modèles JML, en particulier pour les postconditions, qui font référence à la fois à l'état avant et à l'état après l'exécution d'une méthode.

Ainsi, le mot-clé `\old(x)` permet de faire référence à la valeur de l'expression `x` à l'état d'avant. Il est à noter que l'expression `x` peut-être aussi bien un attribut, qu'un appel de méthode, ou encore qu'une expression plus complète.

`\not_modified(x)` est une expression booléenne qui indique que `x` n'a pas été modifié entre l'état avant et l'état après. Cette expression se réécrit suivant deux cas de figure : (*i*) si `x` est une valeur d'un type prédéfini (entier, caractère, etc.), alors elle signifie `x == \old(x)`, (*ii*) si `x` est un objet, alors elle signifie `x.equals(\old(x))`.

On trouve également dans les postconditions l'opérateur `\fresh(o)` qui permet d'exprimer qu'un objet `o` a été nouvellement alloué.

Les opérateurs spécifiques aux objets

JML introduit des opérateurs de manipulation de types dynamiques d'objets évitant l'utilisation des classes réflexives de Java –le package `java.lang.reflect`– pour faire référence au type d'une instance. Ainsi, les types dynamiques des objets sont de type `\TYPE`.

L'opérateur `\type(C)` permet d'introduire un littéral de type `\TYPE` qui désigne le type décrit par la spécification de la classe `C`.

Pour connaître le type dynamique d'un objet `o`, on utilise l'opérateur `\typeof(o)` qui retourne une expression de type `\TYPE`.

L'opérateur `<:` permet de savoir si un type, i.e., une expression de type `\TYPE` est un sous-type d'un autre.

Par exemple, vérifier que le type dynamique d'un objet `o` est un sous-type de la classe `Personne` est exprimé par l'expression JML : `\typeof(o) <: \type(Personne)`

4.1.2 Quelques concepts de JML

JML se situe au même niveau d'abstraction que Java. De ce fait, les spécifications JML portent sur les même attributs que les classes Java et les méthodes partagent la même signature. Les créateurs de JML ont désiré proposer un langage qui soit aisément compréhensible et utilisable par des développeurs Java. De ce fait, les annotations sont embarquées au sein même des classes décrites dans le programme. Mais cette concordance va plus loin. La syntaxe des expressions, et notamment des prédicats JML est basée sur la syntaxe Java.

Variables abstraites

JML donne la possibilité de déclarer des *attributs abstraits*, ne servant qu'en interne au modèle. Ces attributs sont déclarés à l'intérieur d'un bloc JML, et sont précédés d'un modificateur spécifique de JML : `model` ou `ghost`.

Les attributs abstraits `model` sont liés à des attributs de classe, dits concrets. La relation entre un attribut abstrait et un attribut concret est précisée dans une clause `represents`, comme illustré dans l'exemple ci-après. En revanche, les attributs `ghost` n'ont aucune obligation d'être liés à un attribut concret.

Exemple 4.1 (Attributs `model` et `ghost`) Considérons une classe C, contenant un attribut de type *ArrayList* nommé `list` et un attribut de modèle de type entier et nommé `size`. Les déclarations de ces variables s'effectuent de la manière suivante :

```
class C {
   ArrayList list;

   //@ model int size;
   //@ represents size <- list.size();

   //@ ghost boolean isChecked;
   ...
}
```

Cet exemple nous permet d'illustrer un des principes de fonctionnement de JML qui est l'utilisation de clauses contenant les éléments de la modélisation. Ici la clause `represents` permettant d'associer un attribut `model` à une expression.

Utilisation de clauses de spécifications

La modélisation en JML utilise des *clauses* pour décrire la modélisation du système. Ces clauses sont de deux types. On distingue les clauses s'appliquant aux classes, appelées *clauses de spécification de types*, telles que : la clause `initially`, la clause `invariant`, ou la clause `constraints`, décrivant la partie statique du modèle. Elles s'opposent aux

clauses s'appliquant à la partie dynamique, appelées *clauses de spécification de méthodes* portant sur les méthodes, telles que la clause `requires`, la clause `diverges`, la clause `assignable`, la clause `ensures`, ou la clause `signals`.

Ces clauses sont exprimées à l'aide de prédicats de la logique du premier ordre exprimées dans la syntaxe des prédicats Java/JML. Les deux parties suivantes font l'objet de la description de ces clauses, d'abord les clauses de spécification de types, puis les clauses de spécification de méthodes.

4.2 Spécification de types JML

Les clauses de spécification de types décrivent des propriétés qui portent sur les attributs de classe (concrets ou abstraits). On trouve parmi elles, les contraintes initiales (clause `initially`), les invariants (clause `invariant`), et les contraintes historiques (clause `constraint`). Ces propriétés s'appliquent à tous les objets du type considéré. Ces clauses et leur application dans les classes et les sous-classes font l'objet de cette partie.

4.2.1 Contraintes initiales

Une clause `initially` décrit des propriétés qui doivent être établies à la création d'un objet.

Les contraintes initiales d'un objet sont exprimées à l'aide du mot-clé `initially` suivi d'un prédicat JML. Plusieurs contraintes initiales peuvent être spécifiées à l'aide de plusieurs clauses `initially`. La contrainte initiale résultante est ainsi calculée comme la conjonction de ces contraintes.

Exemple 4.2 (Clause `initially`) Considérons une classe C contenant un attribut entier `max`. Spécifier que la valeur de `max` doit initialement être à 10000 s'exprime de la manière ci-dessous :

```
class C {
   int max = 10000;
   //@ initially max == 10000;
}
```

4.2.2 Invariant

Les invariants de classe permettent d'exprimer des propriétés sur les valeurs des attributs de classe qui doivent être vraies à tous les états dits *visibles* du système. L'état visible d'une classe est défini comme l'état atteint à la fin de l'exécution d'une méthode, ou d'un constructeur qui n'est pas déclaré avec le modificateur `helper`[11].

Un invariant est exprimé à l'aide du mot-clé `invariant` suivi d'un prédicat JML.

[11] Une méthode `helper` autorise l'invariant à ne pas être préservé suite à son exécution.

Exemple 4.3 (Clause `invariant`) Considérons une classe C contenant des attributs entiers `max` et `val`. Spécifier que `val` est toujours positif et inférieur à `max` peut s'exprimer par l'invariant ci-dessous :

```
class C {
   int val;
   int max = 10000;
   //@ invariant val >= 0 && val <= max;
   ...
}
```

4.2.3 Contraintes historiques

Les contraintes historiques représentent des propriétés qui doivent être vraies dans tous les états visibles du système, sur le même principe que les invariants. Néanmoins, les contraintes historiques ne sont pas tenues d'être vérifiées suite à l'invocation d'un constructeur de classe ; l'objet créé n'ayant pas d'état précédent. Ces contraintes historiques, également appelées *invariants dynamiques*, sont exprimées à l'aide de *prédicats avant-après*.

Comme il s'agit de propriétés qui doivent s'appliquer à toutes les méthodes, les prédicats exprimés à l'intérieur des contraintes historiques doivent être *transitifs*, i.e., ils doivent considérer les cas où les attributs de classe ne changent pas de valeur. Par exemple, la décroissance stricte d'un attribut peut être une contrainte trop forte qui doit plutôt s'exprimer sous la forme d'une décroissance non stricte, ou alors sous la forme d'une implication où la garde filtre les cas inintéressants (par exemple, quand l'attribut n'est pas modifié par la méthode).

Exemple 4.4 (Clause `constraint`) Considérons une classe C contenant un attribut, arbitrairement déclaré comme abstrait, `variant` supposé décroître strictement après chaque appel de méthode. Ceci s'exprime de la manière suivante :

```
class C {
   /*@ ghost @*/ int variant;
   //@ constraint \old(variant) > 0 ==> \old(variant) > variant;
   ...
}
```

Nous avons vu les principales clauses de spécification de type. Celles-ci s'appliquent non seulement à la classe dans laquelle elles sont déclarées, mais également sur toutes les sous-classes qui en héritent.

4.2.4 Héritage

Lorsqu'une classe hérite d'une super-classe, elle hérite de ses propriétés. Ainsi, les clauses `invariant`, `constraints` et `initially` sont héritées dans toutes les sous-classes. La sémantique de JML considère de ce fait que les propriétés de la super-classe *renforcent*

la modélisation.

Au cas où des attributs sont redéfinis, la question est de savoir si les attributs mis en jeu dans ces propriétés héritées sont relatives à la super-classe ou à la sous-classe. Néanmoins, les prédicats JML sont supposés être évalués dans le cadre dans lequel ils sont décrits. En cas de redéfinition d'un attribut dans une sous-classe, qui génère des problèmes de liaisons retardées, les attributs référencés dans les clauses de spécification de types héritées sont ceux relatifs à la super-classe. Illustrons ce problème par l'exemple ci-dessous.

Exemple 4.5 (Héritage des propriétés et liaisons dynamiques) Considérons une classe `SuperC` dont hérite une classe `C`. Chacune de ces classes contient un attribut nommé `val` sur lequel s'applique un invariant.

```
class SuperC {

   //@ invariant val >= 0;
   int val;
}

class C extends SuperC {

    //@ invariant val =< 100;
    int val;

    void m() {
       val = -1;
    }
}
```

A la lecture rapide de l'invariant dans la classe `C`, on pourrait être amené à croire que l'invariant résultant forcera `val` à être compris entre `0` et `100`, et que l'exécution de `C::m()` engendrera la violation de l'invariant. Or, ce n'est pas le cas.

La sémantique de JML considère que l'invariant de la super-classe doit être vrai mais il ne s'applique que dans le contexte de la super-classe. Ainsi, la propriété `val >= 0` portera sur l'attribut `val` de la classe `SuperC`. De ce fait, l'invocation de la méthode `m()` sur une instance de `C` ne violera pas l'invariant, car seul l'invariant propre à la classe `C` doit être vérifié sur ces attributs. On remarquera que si l'attribut `val` n'avait pas été redéfini dans la classe `C`, l'exécution de la méthode `m()` aurait violé l'invariant.

Nous avons décrit les clauses de spécification de types et les subtilités liées à leur héritage, voyons maintenant l'expression des spécifications de méthodes.

4.3 Spécification de méthodes JML

Les clauses JML appliquées aux méthodes sont déclarées à l'intérieur de blocs de *spécifications de méthodes*. Celles-ci décrivent des comportements séparés par le mot-clé `also`. Chacun des comportements est composé des clauses, comme illustré dans la figure 4.1.

Dans cette figure, M représente les modificateurs appliqués à la méthode *meth* considérée. Cette méthode retourne une donnée de type T. Prenons la description de ces éléments un par un.

```
/*@ behavior
  @    requires P;
  @    diverges D;
  @    assignable A;
  @    ensures Q;
  @    signals (E_1 e1) S_1;
  @    ...
  @    signals (E_N eN) S_N;
  @*/
M T meth(T_1 p_1, ...) throws E_1, ..., E_N { ... }
```

FIG. 4.1 – Clauses de spécification de méthodes JML

4.3.1 Les clauses JML de spécification de méthodes

Nous décrivons dans cette sous partie les différentes clauses de spécification de méthodes, à savoir : les préconditions, la divergence, les champs modifiés, les postconditions normales et exceptionnelles.

Préconditions La précondition P de la méthode est exprimée par la clause `requires`. Elle représente la condition qui *doit être remplie*, à la fois par l'état du système et les valeurs des paramètres p_i, pour autoriser l'appel à la méthode. La valeur par défaut pour cette clause est `true`.

Divergence La clause `diverges` exprime une condition D sous laquelle la méthode *peut* ne pas terminer, i.e., partir dans une boucle infinie. Il est à noter que si cette clause est vraie, alors il n'y a aucune obligation pour que la méthode ne termine pas. La valeur par défaut pour cette clause est `false`.

Champs modifiés La clause `assignable` donne la liste A des éléments qui sont modifiés par l'exécution de la méthode. Si cette clause permet de connaître précisément quels sont les éléments qui vont être modifiés par l'exécution de la méthode, c'est à la charge de la postcondition de préciser les nouvelles valeurs de chacun de ces éléments. Dans le cas où la postcondition n'en fait pas mention, l'élément peut alors avoir n'importe quelle valeur inhérente à son type.

Pour permettre d'être le plus expressif possible, JML donne la possibilité d'employer des expressions régulières. Ainsi, `//@ assignable F.*;` désigne que tous les champs de l'attribut `F` sont modifiés. Si la clause `assignable` est omise, sa valeur par défaut est `\everything` qui signifie que tous les éléments sont susceptibles d'être modifiés. Cette valeur spéciale s'oppose à `\nothing`, qui précise que la méthode ne modifie aucun attribut.

Postcondition normale La clause `ensures` décrit la postcondition normale Q, c'est-à-dire la postcondition que la méthode *s'engage à établir* lorsque celle-ci termine normalement, i.e., sans déclencher d'exception.

Lorsque plusieurs clauses `ensures` sont présentes, la postcondition de la méthode est définie comme la conjonction de chacun des prédicats de ces clauses.

Postcondition exceptionnelle Les clauses `signals` décrivent les postconditions exceptionnelles, c'est-à-dire les postconditions qui sont établies lorsque la méthode termine en déclenchant une exception. L'exception déclenchée est par ailleurs également exprimée dans cette clause, soit en précisant l'objet héritant de la classe `Exception` qui lui est associé (par exemple `e1` dans la figure 4.1), soit de manière implicite : `//@ signals (`E`)`.

Lorsque plusieurs clauses `signals` sont présentes, la postcondition exceptionnelle peut s'exprimer sous la forme d'une seule clause `signals` telle que :

```
                                    //@ signals (Exception e)
//@  signals (E1) S1;                      (e instanceof E1 ==>  S1) &&
//@  signals (E2) S2;   ⇒                 (e instanceof E2 ==>  S2) &&
//@  signals (E3) S3;                      (e instanceof E3 ==>  S3);
```

Remarque : On peut utiliser dans les postconditions JML le mot-clé `\result` pour désigner la valeur de retour de la méthode (si celle-ci n'est pas de type `void`). Seules les expressions qui sont entourées de `old()` représentent des valeurs à l'état avant. Il est à noter que les paramètres, conformément à la sémantique de Java, ont la même valeur à l'état après qu'à l'état avant. Dans ce cas, seules les références des objets en paramètres ne sont considérées qu'à l'état avant. En revanche, les attributs de ces objets passés en paramètre peuvent très bien être modifiés par l'exécution de la méthode.

4.3.2 Les comportements des méthodes JML

JML distingue différents types de comportements : les comportements normaux et exceptionnels. Néanmoins, comme nous allons le voir, ces comportements peuvent s'exprimer sous la syntaxe d'un seul comportement plus général.

Différents types de comportements

Les comportements des méthodes sont décrits par l'intermédiaire de blocs contenant des clauses JML. Néanmoins ces blocs peuvent être précédés de mot-clés (`normal_behavior`, `exceptional_behavior`, ou `behavior`) indiquant le *type de comportement* (normal, exceptionnel, ou mixte) qui va être explicité.

Comportement normal Un comportement normal précise que si la méthode est appelée dans des conditions satisfaisant les préconditions du comportement, alors elle termine normalement, sans déclencher d'exception. Cet état de fait se signale par le mot-clé `normal_behavior`. La présence d'une clause `signals`, décrivant une postcondition exceptionnelle, dans ce bloc est proscrite.

Comportement exceptionnel Un comportement exceptionnel, signalé par le mot-clé `exceptional_behavior`, indique que si la méthode est appelée dans les conditions

licites du comportement, alors celle-ci termine forcément en déclenchant une exception. L'utilisation d'une clause `ensures`, décrivant une postcondition normale, est interdite.

Comportement mixte Un comportement "mixte" permet d'exprimer à la fois une terminaison normale, et une terminaison exceptionnelle.

Les comportements normaux et exceptionnels peuvent se réécrire sous l'expression d'un comportement mixte, dans laquelle une des clauses de postcondition (normale ou exceptionnelle selon le cas) est à la valeur `false`, comme illustré en figure 4.2.

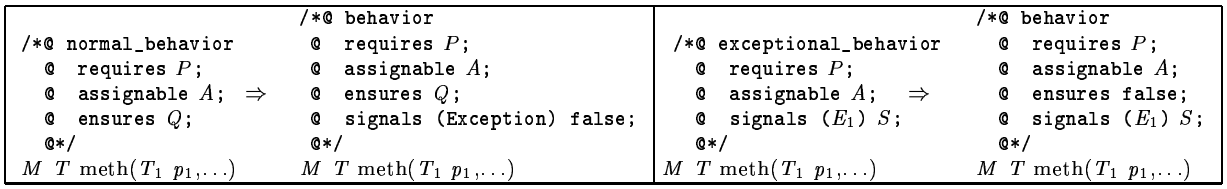


FIG. 4.2 – Réécriture des comportements JML sous une forme générale

La partie droite (resp. gauche) de cette figure illustre la réécriture d'un comportement normal (resp. exceptionnel) en un comportement mixte, en introduisant une clause `signals` (resp. `ensures`) ayant pour valeur `false`, signifiant que le comportement décrit ne peut pas terminer exceptionnellement (resp. normalement).

Réécriture des blocs de comportements

Lorsque la spécification d'une méthode en JML est découpée en plusieurs blocs de spécification de méthode séparés par le mot-clé `also`, JML considère une réécriture de ces différents blocs en un seul, selon le principe illustré en figure 4.3.

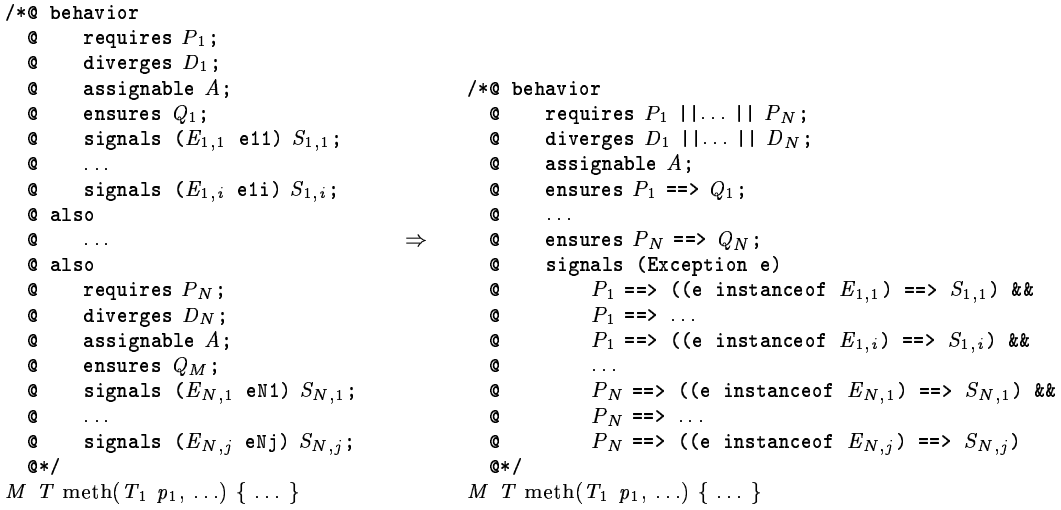


FIG. 4.3 – Réécriture des blocs de spécification de méthodes

La précondition réécrite se calcule comme la disjonction de chacune des préconditions des différents blocs. Il en va de même pour la clause de divergence. La clause `assignable`

doit être identique dans tous les blocs de spécification. Pour ce faire, chaque bloc doit être écrit pour prendre en compte toutes les variables modifiables par chacun des blocs (en précisant leur valeur après). Cette réécriture est expliquée dans [RL00]. Dans la suite du document, nous considérerons que nous disposons d'un moyen d'appliquer cette réécriture, et, par conséquent, les clause `assignable` des différentes spécifications de méthodes JML ne seront pas toujours identiques. La postcondition normale résultante est la conjonction de chacune des postconditions des blocs, sous l'hypothèse que la précondition du bloc soit respectée. Les postconditions exceptionnelles sont réécrites sur le même principe, en considérant une super-classe d'exception qui se décline en différentes sous-classe, chacune établissant une postcondition exceptionnelle.

Comme le laisse supposer cette figure, JML autorise l'écriture de comportements non-exclusifs, considérant que si plusieurs comportements peuvent être activés –i.e., si leurs préconditions sont remplies– alors chacune des postconditions du comportement sera établie. La question de savoir si la spécification de méthode doit être déterministe ou non est laissée à la discrétion des spécifieurs et des outils qui travaillent avec JML.

4.3.3 Notion de pureté

Les méthodes qui ne modifient pas les attributs sont dites *pures* et peuvent ainsi être utilisées dans les prédicats JML. Par conséquent, pour autoriser la présence d'un appel de méthode dans les prédicats de modélisation JML, que ce soit dans les clauses de types ou dans les clauses de méthodes, la méthode considérée doit être déclarée à l'aide du modificateur JML `pure`.

Exemple 4.6 (Illustration d'une méthode `pure`) Considérons une classe C qui contient un attribut `val`, et une méthode *pure* permettant de récupérer la valeur de cet attribut.

```
class C {

  int val;

  //@ invariant getVal() >= 0;

  /*@  assignable \nothing;
    @  ensures \result == val;
    @*/
  public /*@ pure */ int getVal() { return val; }
}
```

Pour comprendre pourquoi cette restriction existe, il faut remonter aux fondements de JML. JML a été conçu pour permettre l'ajout d'assertions qui visent à être vérifiées à la volée lors de l'exécution du programme. Or, ajouter la vérification de ces assertions au programme ne doit pas changer le comportement de celui-ci. En particulier, les états du système, avec et sans vérification d'assertions, doivent être les mêmes. Ainsi, il n'est

pas envisageable que les méthodes invoquées dans les prédicats JML changent l'état du système. Ce principe de fonctionnement sera détaillé en partie 4.5.2 lors de la description du Runtime Assertion Checker.

4.3.4 Spécification de sous-types

Lorsqu'une classe hérite d'une super-classe, elle hérite de ses méthodes, et de leurs spécifications. Les méthodes qui sont surchargées connaissent un traitement différent : suivant le principe de la conception par contrat, les préconditions de la méthode de la sous-classe sont affaiblies et ses postconditions sont renforcées.

La sémantique de JML considère que les spécifications de méthode sont héritées. La spécification de méthode de la sous-classe commence toujours par un `also` suivi de la description des comportements de la méthode.

Ainsi, en appliquant les réécritures on constate que les méthodes surchargées dans une sous-classe autorisent les mêmes comportements que dans la super-classe, plus des comportements supplémentaires. La précondition résultante est donc *affaiblie* (à cause de la disjonction des préconditions) puisqu'elle autorise de plus larges possibilités d'exécution, mais la postcondition est *renforcée* (à cause des postconditions supplémentaires).

Exemple 4.7 (Héritage des spécifications de méthodes et liaisons dynamiques) Considérons à nouveau les classes `SuperC` et `C` de l'exemple précédent sur l'héritage. Nous rajoutons dans la classe `SuperC` une méthode `me(int)` qui est surchargée dans la sous-classe.

```
class SuperC {

  //@ invariant val >= 0;
  int val;


  /*@ requires i >= 0 && i <= 80;
    @ assignable val;
    @ ensures val == i;
    @*/
  public void me(int i) { ... }
```

```
class C extends SuperC {

  //@ invariant val <= 100;
  int val;

  /*@ also
    @   requires i >= 30 && i <= 300;
    @   assignable val;
    @   ensures val == 90;
    @*/
  public void me(int i) { ... }
```

En appliquant les règles de réécriture des `also` nous obtenons trois comportements distincts pour l'invocation de `C::me(int)` : (*i*) lorsque la précondition de `SuperC::me(int)` est satisfaite mais pas celle de `C::me(int)` (`i >= 0 && i < 30`), (*ii*) lorsque la précondition de `C::me(int)` est satisfaite mais pas celle de la classe `SuperC::me(int)` (`i > 80 && i <= 300`), et (*iii*) lorsque ces deux préconditions sont satisfaites (`i >= 30 && i <= 80`).

Dans ce dernier cas, on pourrait être tenté de croire que la postcondition est inconsistante puisqu'il est impossible que la valeur de `val`, dans la postcondition, puisse être à la fois égale à 90 et comprise entre 30 et 80.

Néanmoins, il faut garder à l'esprit que le comportement, tout comme les propriétés de classe, s'applique dans le contexte de la classe dans laquelle il est déclaré. Ainsi la

postcondition `val == i` considère l'attribut `val` déclaré dans la classe `SuperC`, tandis que la postcondition `val == 90` considère l'attribut `val` déclaré dans la classe `C`.

Après avoir présenté les mots-clés JML, permettant la spécification de types et de méthodes, nous introduisons à présent un exemple de modèle JML qui sera utilisé tout au long de ce mémoire.

4.4 Exemple fil rouge

À l'heure actuelle, la principale utilisation de JML est d'annoter des programmes Java Card [Sun00]. L'exemple choisi est inspiré de ce genre de programmes, et décrit un porte-monnaie électronique. Ce porte-monnaie se décline en deux versions : une version standard et une version dans laquelle le montant des transactions est limité.

4.4.1 Description

Le porte-monnaie électronique que nous décrivons permet de stocker une certaine somme d'argent. Il est impératif que le solde du porte-monnaie soit toujours positif. Ainsi toutes les dépenses et transactions doivent s'assurer de ne pas obtenir un solde débiteur.

Initialement, le solde du porte-monnaie est égal à zéro. Il est ensuite crédité à l'aide d'une méthode nommée `crédit`. Lorsqu'un paiement est demandé, deux cas de figure se présentent : soit le montant à débiter est licite (i.e., il est positif et il y a suffisamment d'argent pour effectuer le paiement) et le débit sur le porte-monnaie a lieu, soit il est illicite (si le paramètre est négatif ou trop élevé), et dans ce cas, une exception est déclenchée. Il doit également être possible de transférer de l'argent d'un porte-monnaie vers un autre. Une méthode d'observation doit permettre de connaître le solde du porte-monnaie.

Chaque transaction modifiant le solde du porte-monnaie doit être enregistrée dans un historique. Il doit être possible d'annuler la dernière transaction pour revenir au solde précédent.

Une version étendue de ce porte-monnaie restreint le solde à un maximum fixé initialement à 100 euros. Par conséquent, l'opération de crédit doit également prendre en compte cette limitation. Dans le cas où le crédit ne peut avoir lieu, la transaction est annulée et une exception est déclenchée. Il doit également être possible de modifier le plafond associé au porte-monnaie.

Le modèle que nous proposons est décrit ci-après.

4.4.2 Modélisation Java/JML

L'exemple va considérer quatre classes :

- Une classe `Purse` qui décrit le fonctionnement du porte-monnaie standard ;
- Une classe `History` qui permet de stocker l'historique du solde du porte-monnaie ;
- Une classe `LimitedPurse` qui décrit le fonctionnement du porte-monnaie limité ;
- Une classe d'exception `WrongParameterException` qui informe que les paramètres saisis sont incorrects.

L'exception `WrongParameterException`

Nous donnons dès à présent la description de la classe d'exception. Nous ne nous attarderons pas sur celle-ci, car elle ne contient pas de modélisation JML ; celle-ci n'apportant pas d'éléments intéressants du point de vue des traitements que nous allons effectuer.

```
class WrongParameterException extends Exception {

    public WrongParameterException() { ... }

}
```

La classe `History`

Cette classe permet de stocker les valeurs successives du solde du porte-monnaie. Nous considérons une structure de liste chaînée pour permettre de remonter dans l'historique autant que nécessaire. Elle présente deux attributs : l'un pour stocker le solde et l'autre pour donner un accès sur l'historique précédente. On dispose de deux accesseurs permettant de récupérer les valeurs de ces deux attributs. La classe `History` se présente de la manière suivante :

```
class History {

    //@ invariant bal >= 0;
    short bal;

    History prev;

    /*@ behavior
      @   requires b >= 0;
      @   assignable bal, prev;
      @   ensures bal == b && prev == p;
      @*/
    public History(short b, History p) { ... }

    /*@ behavior
      @    assignable \nothing;
      @    ensures \result == bal;
      @*/
    public /*@ pure */ short getBalance() { ... }

    /*@ behavior
      @    assignable \nothing;
      @    ensures \result == prev;
      @*/
    public /*@ pure */ History getPrevious() { ... }

}
```

La classe `Purse`

Intéressons nous désormais à la classe `Purse`. Cette classe contient un champ entier `balance` qui représente le solde du porte-monnaie exprimé en centimes d'euros. L'invariant JML spécifie que ce champ n'est jamais négatif et le constructeur de la classe instancie ce champ à un entier court positif passé en paramètre.

```
class Purse {

   //@ invariant balance >= 0;
   short balance;

   History hist;

   //@ initially hist == null;

   /*@ normal_behavior
     @   requires b >= 0;
     @   assignable balance;
     @   ensures balance == b && hist == null;
     @*/
   public Purse(short b) { ... }

   ...

}
```

La méthode `credit` permet de créditer le porte-monnaie d'une certaine somme d'argent. Pour sauvegarder la valeur modifiée par le crédit, on créé une nouvelle entrée dans l'historique stockant l'ancienne valeur du solde, avant le crédit.

```
   /*@ normal_behavior
     @   requires a > 0;
     @   assignable balance, hist;
     @   ensures balance == (short)(\old(balance) + a);
     @   ensures \fresh(hist) && hist.getBalance() == \old(balance)
     @           && hist.getPrevious() == \old(hist);
     @*/
   public void credit(short a);
```

La méthode de débit contient deux comportements permettant de terminer soit normalement par le débit du montant souhaité, soit de déclencher une exception de type `WrongParameterException`. Si le débit a lieu, on créé une nouvelle valeur d'historique stockant la valeur du solde avant le débit.

```
   /*@ normal_behavior
     @   requires a > 0 && a <= balance;
     @   assignable balance, hist;
     @   ensures balance == (short)(\old(balance) - a);
     @   ensures \fresh(hist) && hist.getBalance() == \old(balance)
     @           && hist.getPrevious() == \old(hist);
     @ also
     @   exceptional_behavior
     @     requires a =< 0 || a > balance;
     @     assignable balance
     @     signals (WrongParameterException) balance == \old(balance) &&
     @                                       hist == \old(hist);
     @*/
   public void debit(short a) throws WrongParameterException { ... }
```

La méthode d'observation suivante permet de connaître le montant du solde du porte-monnaie.

```
  /*@ normal_behavior
    @   assignable \nothing;
    @   ensures \result == balance;
    @*/
  public /*@ pure */ short getBalance() { ... }
```

La méthode de transfert permet de basculer le contenu d'un porte-monnaie dans un autre. Le transfert écrase le solde du porte-monnaie courant avec le solde du porte-monnaie passé en paramètre.

```
/*@ normal_behavior
  @   requires p != null;
  @   assignable balance, hist;
  @   ensures balance == \old(p.balance);
  @   ensures \fresh(hist) && hist.getBalance() == \old(balance) &&
  @           hist.getPrevious() == \old(hist);
  @*/
public void transfer(Purse p);
```

La dernière méthode de cette classe permet d'annuler la dernière opération qui a été effectuée, utilisant l'historique pour restaurer l'ancienne valeur sauvegardée.

```
/*@ normal_behavior
  @   requires hist != null;
  @   assignable hist, balance;
  @   ensures balance == \old(hist.getBalance());
  @   ensures hist == \old(hist.getPrevious());
  @*/
public void cancel();
```

La classe `LimitedPurse`

La classe `LimitedPurse` est une extension de la classe `Purse`. Elle contient un champ `max` qui spécifie la valeur maximale qui peut être contenue dans le porte-monnaie. Cette valeur sera stockée dans une constante commune à toutes les instances de porte-monnaie limité. Par extension, un invariant se rajoute pour majorer la valeur du champ `balance` hérité.

Le constructeur de cette classe est similaire à celui de la classe étendue.

```
class LimitedPurse extends Purse {

  short max = 10000;

  //@ invariant balance <= max;

  //@ initially max == 10000;

  /*@ normal_behavior
    @   requires b >= 0 && b <= max;
    @   assignable balance, hist;
    @   ensures balance == b && hist == null;
    @*/
  public LimitedPurse(short b) { ... }

  ...

}
```

La méthode de crédit est alors redéfinie pour prendre en compte la limitation : on ne peut pas créditer une valeur qui va forcer le solde à dépasser le plafond fixé.

```
/*@ also
  @   normal_behavior
  @     requires a > 0 &&  (short)(balance + a) <= max;
  @     assignable balance;
  @     ensures balance == \old(balance) + a;
  @*/
public void credit(short a) { ... }
```

Cette modélisation est incorrecte. En effet, le concept d'héritage en JML autorise cette méthode à activer les comportements normaux de la super-classe `Purse`, indépendamment de ceux décrits dans la sous-classe. Il faut donc trouver un moyen de n'autoriser que l'un ou l'autre des comportements. Pour cela, nous rajoutons la contrainte `\typeof(this) == \type(Purse)` (resp. `\typeof(this) == \type(LimitedPurse)`) dans la spécification de la méthode `credit(short)` de la classe `Purse` (resp. `LimitedPurse`).

Pour finir, nous autorisons le changement de plafond. Ceci est réalisé grâce à la méthode suivante :

```
/*@ behavior
  @   requires m >= 0 && m >= balance;
  @   assignable max;
  @   ensures max == m;
  @*/
public void setMax(short m) { ... }
```

Comme nous l'avons vu cet exemple illustre les possibilités d'expression de JML, exprimant des spécifications de type (les clauses `initially` et `invariant`) et les clauses de spécification de méthodes (les clauses `requires`, `assignable`, `ensures`, `signals`). Cet exemple sera utilisé pour illustrer l'animation et la génération de tests dans les chapitres suivants.

4.5 Les outils pour JML

Cette partie présente les outils associés à JML, en complétant les explications données en partie 2.2 de l'état de l'art. Nous nous focalisons uniquement sur les outils fournis avec les distributions de JML, disponibles sur le site `http://jmlspecs.org`.

Les outils que nous présentons sont le Runtime Assertion Checker et l'outil de construction de tests unitaires JMLUnit, adaptation de JUnit [GB98] à JML.

4.5.1 Le JML Runtime Assertion Checker

Le Runtime Assertion Checker (RAC) [CL02a] a été développé par Gary T. Leavens et Yoonsik Cheon à l'université de l'Iowa. Il s'agit de l'outil de base de JML. Il se présente sous la forme d'un compilateur qui permet de produire un code Java enrichi par la vérification, lors de l'exécution des méthodes, des différentes annotations JML.

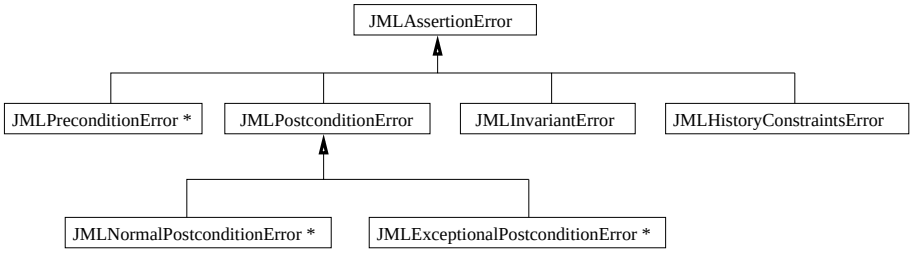


FIG. 4.4 – La hiérarchie des exceptions du Runtime Assertion Checker

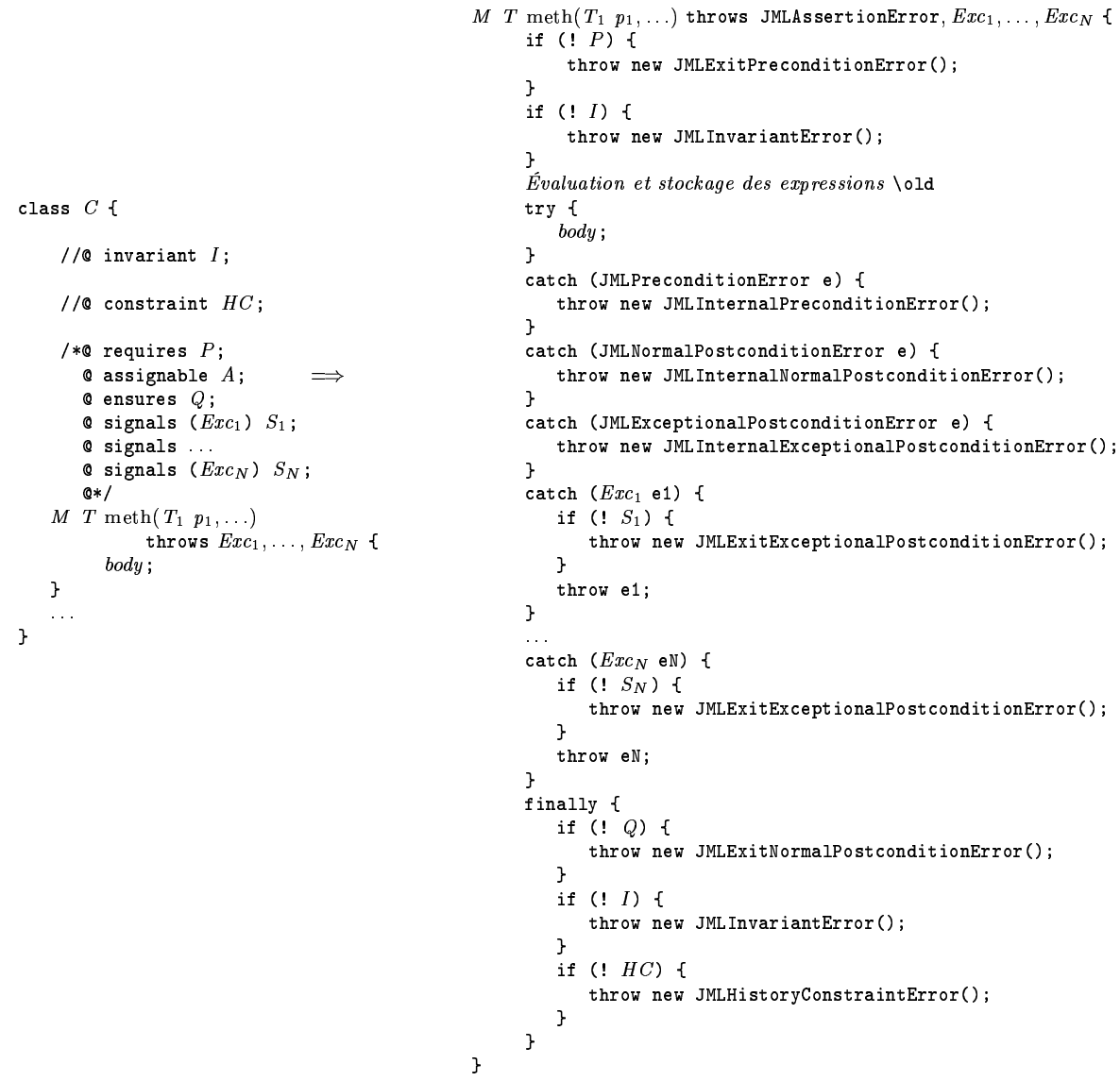


FIG. 4.5 – Principe de fonctionnement du Runtime Assertion Checker

Cet outil utilise le déclenchement d'exceptions spécifiques, signalant la violation d'une des assertions JML. Chacune de ces exceptions contient les informations de retour à l'utilisateur concernant l'assertion incriminée dans le code source, et l'état du système et des paramètres lorsque celle-ci est déclenchée. La hiérarchie des exceptions du RAC est donnée par la figure 4.4. Dans cette figure, les classes d'exceptions suffixées par une astérisque (*) se décomposent en deux classes spécialisées, représentant le fait que la violation de l'assertion est faite dans la méthode exécutée (par ex. `JMLExitPreconditionError` spécialisant les `JMLPreconditionError`) ou dans une des méthodes exécutées (par ex. `JMLInternalPreconditionError`, autre spécialisation de `JMLPreconditionError`).

Le principe de fonctionnement de ce compilateur est de rajouter du code, correspondant à la vérification des assertions JML, dans le code source des classes considérées. Si les assertions ne sont pas vérifiées, des exceptions de type `JMLAssertionError` sont dé-

clenchées, comme illustré en figure 4.5. Dans cette figure, le code Java/JML placé sur la droite produit l'équivalent du byte-code Java sur la gauche, en procédant de la manière suivante. Au début de l'exécution de la méthode, les préconditions et l'invariant sont vérifiées. Les valeurs des expressions `\old` sont ensuite évaluées et stockées pour permettre la vérification ultérieure des postconditions utilisant ces valeurs. Si l'exécution du corps de la méthode déclenche une exception, deux cas se présentent :

1. l'exception récupérée est une exception JML (`JMLPreconditionError`, `JMLNormalPostconditionError`, `JMLExceptionalPostcondition`) : le programme déclenche une version raffinée interne de l'exception récupérée.
2. l'exception récupérée est une exception attendue par le programme : les postconditions exceptionnelles relatives à l'exception spécifiée sont alors vérifiées et l'exception récupérée est ensuite re-déclenchée.

Pour finir, les postconditions normales sont vérifiées, ainsi que l'invariant de classe et les contraintes historiques.

Le compilateur `jmlc` produit le byte-code correspondant au code Java enrichi des vérifications d'assertions JML. L'exécution de ce byte-code permettra donc la levée des exceptions si cette exécution viole une des assertions JML. On notera toutefois l'absence de vérification de la clause `assignable`.

4.5.2 JMLUnit

JMLUnit est la combinaison de JUnit et du Runtime Assertion Checker décrit précédemment. Cette combinaison a été introduite dans la continuité du RAC, par Y. Cheon et G. T. Leavens [CL02b]. Elle consiste à exécuter des tests en utilisant les possibilités de JUnit, sur des classes Java enrichies par le Runtime Assertion Checker.

JUnit

JUnit [GB98] est un outil d'exécution de tests unitaires sur du code Java. JUnit permet de produire un ensemble de classes d'objet permettant la définition, le passage de cas de test et l'établissement d'un verdict pour les tests unitaires de méthodes Java.

Illustrons l'utilisation de JUnit par un exemple, en considérant l'exemple de classe JUnit donné en figure 4.6. Cette figure décrit une classe JUnit qui sert de cas de test. Cette classe hérite de la classe `TestCase` fournie par l'environnement JUnit. Cette classe représente un cas de test, identifié par un nom passé sous la forme d'une chaîne de caractères dans le constructeur de la classe. Nous allons maintenant décrire les méthodes spécifiques à JUnit.

Pour commencer, la méthode `setUp()` sert à décrire un préambule permettant de placer l'objet sous test dans un état intéressant. La méthode `tearDown()`, à l'inverse, contient le postambule qui s'exécute lorsque le test est terminé. Les méthodes préfixées par *test* sont les méthodes qui vont être exécutées pendant le corps du test. Par convention, ces méthodes ne prennent aucun argument et ne retournent aucune valeur. Dans notre exemple, nous avons deux méthodes de tests : `testTransfer` et `testDebit`.

Une suite de tests est construite à partir des méthodes de test, par l'intermédiaire d'un objet TestSuite, construit par inspection des méthodes de la classe Java spécifiée préfixées par *test*.

La classe `TestCase` contient une méthode `runTest()` qui est en charge automatiquement d'exécuter la suite de tests définie, de vérifier l'état obtenu par rapport à l'oracle et de créer des objets `TestResult` contenant les résultats du passage des tests. Ainsi l'oracle doit être spécifié manuellement par l'utilisateur de JUnit.

L'utilisation de JML apporte des changements du point de vue du calcul de l'oracle, comme nous allons le voir à présent.

JML + JUnit = JMLUnit

Le modèle JML, une fois traduit en assertions Java et inséré à l'intérieur du code, fournit un oracle partiel aux tests qui sont produits. Par partiel, nous désignons le fait que cet oracle ne va pas permettre de vérifier de manière exhaustive deux états d'un système objet (en comparant chacun des attributs des objets deux-à-deux). Cette difficulté est due au fait que les attributs composant un système objet ne sont pas toujours visibles/accessibles.

Passer par JML et la vérification des assertions à la volée est donc un bon moyen de contourner le problème de l'oracle. Ainsi, lorsqu'aucune assertion JML n'est violée lors du passage du cas de test, le test réussit. Dans le cas contraire, le test échoue. Cette dernière affirmation est à nuancer, car la récupération d'une exception `JMLEntryPrecondition-Exception` signale que le test a été exécuté dans un état ne respectant pas le contrat de la méthode testée. Ces tests sont donc isolés et considérées comme non concluants (inconclusive).

En plus d'un oracle direct et immédiat, JMLUnit produit automatiquement, à partir

```
import junit.framework.*;

class PurseTest extends TestCase {

  private Purse p;

  public PurseTest(String name) {
    super(name);
  }

  public void testDebit() {
    p.credit(100);
    p.debit(101);
  }

  public void testTranfer() {
    Purse p2 = new Purse();
    p2.credit(100);
    p.transfer(p2);
  }

  protected void setUp() {
    p = new Purse();
  }

  protected void tearDown() {
  }

  public static Test suite() {
    return new TestSuite(PurseTest.class);
  }

  public static void main(String[] args) {
    String[] testCaseName =
                    {Purse.class.getName()};
    junit.textui.TestRunner
                      .main(testCaseName);
  }
}
```

FIG. 4.6 – Un exemple de classe JUnit

d'une classe Java annotée en JML, les classes d'objets représentant les différents cas de test pour chacune des méthodes contenues dans chacune des classes. JMLUnit génère en plus de cela des listes *prêtes-à-remplir* représentant les données de test qui vont servir de paramètres aux méthodes à tester.

Malheureusement, les données de test et les séquences de test restent encore à la charge de l'utilisateur. Des automatisations ont été mises au point pour alléger ce travail, soit par des méthodes aléatoires (comme Jartege [Ori05]), soit par des méthodes combinatoires (comme Tobias [LdBMB04]). Si ces approches fournissent un très grand nombre de cas de test à moindre coût, les tests produits ne sont pas toujours pertinents, car redondants, et beaucoup de tests insignifiants sont produits pour être par la suite filtrés lors de l'exécution. Ceci est dû au fait que les valeurs choisies ou définies pour les paramètres des méthodes ne sont pas toujours compatibles avec les valeurs requises dans le modèle. Le travail présenté dans cette thèse vise à résoudre ce problème, en employant une approche basée sur le modèle pour produire les cibles de test et les données de test pertinentes.

4.6 Synthèse

Ce chapitre a présenté le Java Modeling Language, langage de modélisation dédié à Java. Comme nous l'avons vu, il existe deux utilisations possibles du JML. La première est le renforcement du code par des assertions qui pourront être vérifiées à la volée lors de l'exécution du programme annoté. La seconde est de considérer ce langage comme une opportunité de spécifier un modèle orienté-objet complet, sans obligation de fournir une implantation pour le corps des méthodes des classes. C'est cette seconde direction qui est le point de départ des contributions de cette thèse.

Comme nous l'avons vu au chapitre précédent, la technologie BZ-Testing-Tools et son format intermédiaire, le BZP, ne sont ni orientés-objet, ni même modulaire. La difficulté majeure consiste alors à exprimer des notions d'objet dans un langage ne permettant de raisonner que sur des ensembles, des relations, des fonctions et des valeurs numériques.

Les contributions exposées dans la partie suivante commencent par expliquer comment exprimer un sous-ensemble intéressant de JML dans le format BZP, ainsi que les changements apportés au format BZP.

Deuxième partie

Contributions

Chapitre 5

Représentation ensembliste des structures de classes Java

Sommaire

Nous présentons dans ce chapitre l'expression d'une structure de classe Java dans le format BZP. Comme nous l'avons vu au chapitre 3, le format BZP est une notation basée sur la définition d'ensembles et la logique du premier ordre avec des constructions ensemblistes et relationnelles.

Notre objectif est d'utiliser les fonctionnalités offertes par le langage BZP pour permettre d'exprimer les notions d'objets de Java, telles que la notion de classes, d'instances, d'attributs et de méthodes. Nous nous attarderons particulièrement sur la prise en compte de l'héritage qui impacte la représentation de ces notions.

Ce chapitre débute par une discussion sur la couverture des structures Java et les objectifs que nous souhaitons atteindre pour réaliser l'animation des modèles JML. Nous abordons ensuite l'expression en BZP des classes, puis des méthodes, avant de finir par la représentation du constructeur de classes par défaut.

5.1 Avant-propos

Notre objectif est d'exprimer dans le format BZP un ensemble de classes Java. D'une part, nous avons un ensemble de classes Java, contenant des attributs et des méthodes. Nous disposons d'éventuelles relations d'héritage entre les classes que nous considérons. D'autre part, nous avons le langage BZP qui permet d'exprimer des modules, des constantes, des variables, et de décrire des opérations à l'aide de prédicats écrits en logique du premier ordre, en manipulant des structures de données énumérées et finies.

Cette sous-partie présente dans un premier temps les restrictions posées sur le langage Java que nous considérons. Elle décrit ensuite les objectifs de l'animation des spécifications JML qui vont guider la modélisation BZP des classes Java.

5.1.1 Restrictions sur le Java traité

Nous décrivons dans cette partie les restrictions que nous posons sur le langage Java supporté. Celles-ci portent sur la manière de modéliser les classes Java et les expressions ou constructions non supportées.

- Les types de données supportés sont les entiers et les types objets. Les tableaux, les entiers longs ou les réels (double ou float) ne sont pas supportés par notre approche, faute de solveur adéquat pour réaliser leur traitement.
- Seuls les attributs numériques peuvent être initialisés lors de leur déclaration dans une classe. Les attributs de classe objets devront être instanciés par un constructeur (pour les attributs d'instance) ou une méthode spécifique (pour les attributs statiques).
- L'utilisation de classes imbriquées (déclaration d'une classe à l'intérieur d'une autre classe) n'est pas autorisée.
- Nous ne modélisons pas le mécanisme de *garbage collector* supposé libérer les adresses occupées par des objets qui ne sont plus liés à aucune référence.
- Nous considérons que les valeurs passées en paramètres aux différentes méthodes sont soit des valeurs numériques soit des instances d'objets auxquelles une référence existe à l'état du système considéré.

5.1.2 Définition des objectifs

Pour réaliser l'animation des spécifications Java/JML, nous avons besoin d'exprimer :

- les états du système, qui donnent une représentation mémoire de l'état du programme principal Java en cours d'exécution ;
- les transitions du système, qui permettent de passer d'un état à un autre, que ce soit en créant un objet ou en invoquant une méthode sur un objet déjà créé.

Pour cela, nous allons exprimer les états du système à l'aide de structures ensemblistes et numériques sur lesquelles peuvent porter des contraintes. Celles-ci seront traitées avec une approche en PLC. Il nous faut considérer des états du système comme un ensemble d'objets portant des attributs et sur lesquels peuvent s'exécuter des méthodes. Chaque objet existant en mémoire possède un certain nombre de caractéristiques :

- une adresse en mémoire ;
- une classe associée qui représente son type dynamique ;
- une valeur pour chacun des attributs ;
- un ensemble de méthodes qu'il peut activer.

Le format BZP impose l'utilisation de structures de données finies. Les types de données dérivés des entiers sont tous finis. La seule possibilité d'introduire des données infinies se situe au niveau des instances d'objets. Dans la pratique, le nombre d'instances en activité est lié à la mémoire du système. Nous choisissons donc de borner la mémoire de manière à n'avoir qu'un nombre fixé d'instances d'objets.

Nous nous fixons deux règles concernant le typage des objets manipulés durant l'animation symbolique des spécifications JML :

1. Nous partons du principe que les valeurs des adresses des objets sont toujours connues, car cela ne présente pas d'intérêt de laisser du non-déterminisme dans l'allocation de l'adresse d'une instance. Ainsi, les valeurs contraintes sont uniquement posées sur les attributs.
2. Le type dynamique d'un objet doit toujours être connu pour permettre, lors de l'appel à une de ses méthodes, de déterminer quelle méthode doit être exécutée. Ainsi dans le cas où un objet pourra avoir plusieurs types dynamiques, à l'issue de l'analyse statique, un point de choix entre ces différents types sera posé. Nous pensons que ce fait peut être d'une certaine aide pour mettre au point un modèle JML non ambigu dans l'objectif de générer des tests à partir de ce modèle.

L'objectif des parties suivantes est d'expliciter la traduction effectuée, ainsi que les choix réalisés. Nous commencerons par l'expression des notions d'objets et d'attributs de classe. Nous aborderons ensuite l'expression des méthodes Java, en se préoccupant uniquement de la traduction des signatures des méthodes d'une classe. La traduction des spécifications JML des classes et des méthodes fera l'objet du chapitre 6.

5.2 Représentation des états

Cette partie s'intéresse à la traduction des éléments relatifs à un état d'un programme Java. L'état d'un programme est caractérisé par un ensemble d'objets, chacun d'eux ayant des attributs qui présentent différentes valeurs. Cette partie définit une modélisation au format BZP d'un état de programme Java.

Cette modélisation se base sur une abstraction des notions objets (adresses mémoires, instances d'objets, etc.) avec une représentation ensembliste. Elle considère un ensemble

fini de classes, présentant éventuellement des relations d'héritage entre elles.

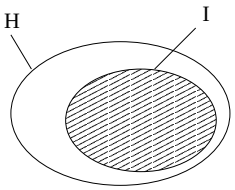
Nous commençons par expliquer l'abstraction réalisée pour représenter les instances d'objets présentes dans le programme. Nous introduisons par la suite la représentation des attributs de chacun des objets.

5.2.1 Représentation des instances de classes

Nous décrivons ici les différentes propriétés qui s'appliquent aux instances de classe, du point de vue des adresses mémoire, de l'héritage, de l'accès aux objets et de leur typage dynamique.

Les instances de classes sont considérées comme des adresses de la mémoire. A chaque adresse est associé un objet. Les adresses des objets sont définies dans une constante représentant l'ensemble des instances possibles.

Le nombre d'adresse en mémoire se doit d'être fini pour garantir la vérification de la consistance des contraintes, tel qu'expliqué en partie 3.3. En conséquence, nous considérons une variable ensembliste qui représente toutes les instances qui ont déjà été créées.



Nous caractérisons à présent les instances d'objets existantes dans la mémoire.

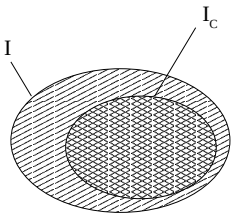
PROPRIÉTÉ 1 (REPRÉSENTATION DES INSTANCES D'OBJETS) *Soit* $H = \{addr_0, addr_1, \ldots, addr_N\}$ *l'ensemble (arbitrairement fini) des adresses initialement libres dans le programme Java. Cet ensemble est appelé le* tas *(heap). Soit I l'ensemble des instances existantes à n'importe quel instant de l'exécution du programme. L'ensemble des instances existantes dans le système est un sous-ensemble des adresses du tas :*

$$I \subseteq H \tag{5.1}$$

Initialement, on a :

$$I = \emptyset \tag{5.2}$$

Nous réalisons une abstraction de l'ensemble des instances d'une classe donnée existantes par un ensemble d'adresses, sous-ensemble de l'ensemble de toutes les instances existantes (et par extension, sous-ensemble du tas).



PROPRIÉTÉ 2 (REPRÉSENTATION DES INSTANCES DE CLASSE) *Soit* I_C *l'ensemble des instances de la classe C. L'ensemble des instances d'objet d'un type donné est un sous-ensemble des instances créées.*

$$I_C \subseteq I \tag{5.3}$$

Initialement, on a :

$$I_C = \emptyset \tag{5.4}$$

Lorsqu'il existe une relation d'héritage entre deux classes, les instances des sous-classes sont, par définition, des instances de la super-classe.

DÉFINITION 3 (REPRÉSENTATION D'UNE SOUS-CLASSE) *Soit C une classe héritant d'une super-classe SC, la relation d'héritage entre C et SC se dénote par :*

$$C \sqsubseteq SC \tag{5.5}$$

Par la suite, nous considérons la notation $\sqsubseteq_N$ pour désigner le niveau d'héritage entre deux classes. Ainsi, $C \sqsubseteq_1 SC$ signifie que C hérite directement de SC. Par transitivité, on a :

$$C \sqsubseteq_1 C' \wedge C' \sqsubseteq_N C'' \Rightarrow C \sqsubseteq_{N+1} C'' \tag{5.6}$$

Nous pouvons désormais définir les relations qui s'appliquent entre les ensembles d'instances de différentes sous-classes par rapport à leur super-classe directe : ces ensembles d'instances sont inclus dans l'ensemble des instances de la super-classe, et sont disjoints deux à deux.

PROPRIÉTÉ 3 (REPRÉSENTATION DE L'HÉRITAGE) *Soit $C_1, \ldots, C_N$ l'ensemble des N sous-classes de SC telles que $C_i \sqsubseteq_1 SC$ pour $1 \leq i \leq N$. Les instances des sous-classes sont également des instances de la super-classe, mais une instance ne peut pas appartenir à deux sous-classes différentes. Ainsi*

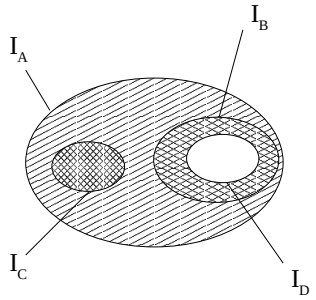
$$\bigwedge_{1 \leq i \leq N} I_{C_i} \subseteq I_{SC} \wedge \bigwedge_{1 \leq i < j \leq N} I_{C_i} \cap I_{C_j} = \emptyset \tag{5.7}$$

Illustrons ces propriétés de la représentation de l'héritage sur un exemple.

Exemple 5.1 (Héritage et instances) Considérons quatre classes A, B, C et D, avec les relations d'héritages suivantes :

$$B \sqsubseteq_1 A, C \sqsubseteq_1 A, D \sqsubseteq_1 B$$

Les contraintes associées à cette hiérarchie expriment l'inclusion de l'ensemble des instances de B et de C dans l'ensemble des instances de A, et l'inclusion de l'ensemble des instances de D dans l'ensemble des instances de B. De plus, les ensembles d'instances de B et de C sont disjoints. Formellement :



$$I_A \subseteq I \wedge I_B \subseteq I_A \wedge I_C \subseteq I_A \wedge (I_B \cap I_C) = \emptyset \wedge I_D \subseteq I_B \tag{5.8}$$

Nous définissons à présent la notion de disponibilité pour un objet. Il s'agit d'une notion importante, car elle est utile pour restreindre les objets susceptibles d'être utilisés en tant que paramètres lors de l'invocation d'une méthode. En effet, il doit être impossible de passer en paramètre un objet qui n'est pas disponible (pour lequel aucun pointeur sur cet objet n'est à disposition).

DÉFINITION 4 (OBJET DISPONIBLE) *Un objet est dit* disponible *s'il existe une référence qui a permis de récupérer l'objet au cours de l'exécution d'une séquence d'animation. Par définition, un objet créé par l'appel à un constructeur est disponible (on suppose qu'on le récupère toujours), et un objet retourné par une méthode l'est également (sous les mêmes hypothèses).*

Tous les objets existants lors de l'exécution d'un programme Java ne sont pas disponibles. Par exemple, les objets créés dynamiquement lors de l'exécution d'une méthode ne le sont pas (à moins d'être en même temps la valeur de retour de la méthode).

PROPRIÉTÉ 4 (RÉFÉRENCES DISPONIBLES) *Soient A (pour "available") l'ensemble des objets disponibles, et I l'ensemble des objets existants. La contrainte suivante s'appliquent entre ces ensembles :*

$$A \subseteq I \cup \{null\} \tag{5.9}$$

Initialement, on a :

$$A = \{null\} \tag{5.10}$$

où null représente l'adresse fictive ne correspondant à aucune adresse mémoire instanciable.

L'ensemble des instances disponibles est une variable destinée à évoluer au cours de l'animation du modèle. Comme nous le verrons en partie 5.3, chaque nouvelle instance créée sera ajoutée à cet ensemble, de même que chaque objet retourné par une méthode.

Nous avons désormais la possibilité d'exprimer le type dynamique d'un objet.

DÉFINITION 5 (TYPE DYNAMIQUE D'UN OBJET) *Le type dynamique d'un objet correspond à la classe contenant le constructeur qui a permis la création de l'objet.*

Pour permettre de connaître le type dynamique d'une instance, nous introduisons une fonction totale associant à chaque instance sa classe.

PROPRIÉTÉ 5 (TYPAGE DYNAMIQUE D'INSTANCES) *Dénotons par T_o le type dynamique d'un objet o. Si C est le type dynamique de o alors o est une des instances de C et n'est instance d'aucune des sous-classes de C de niveau 1 et noté SSC (et par extension, de niveau $n > 1$).*

$$T_o = C \Leftrightarrow o \in I_C \wedge o \notin \bigcup_{SSC \sqsubseteq_1 C} SSC \tag{5.11}$$

Nous avons présenté dans cette sous-partie l'abstraction qui nous permet de modéliser les ensembles d'instances d'un programme Java en prenant en compte l'héritage et la disponibilité des objets. Nous abordons à présent l'expression de cette abstraction dans le format BZP.

5.2.2 Expressions des instances d'objet en BZP

Nous décrivons ici l'expression des concepts décrits précédemment. La gestion des instances de classes est gérée à l'intérieur d'un module nommé `modele`. Celui-ci contient les variables nécessaires à la gestion des instances à savoir :

- la gestion de l'ensemble I des instances existantes exprimé par une variable ensembliste `instances` ;
- la gestion de l'ensemble A des instances disponibles exprimé par une variable ensembliste `available` ;
- le typage dynamique d'un objet T, exprimé par une fonction `typeof`.

Dans cette partie et dans le reste de la traduction, nous exprimerons les propriétés relatives aux états d'un système –c'est-à-dire, les propriétés relatives aux instances de classes, et plus tard le typage des attributs– sous la forme d'invariants BZP.

Par souci de clarté et de lisibilité, le code BZP issu de la modélisation ne sera pas présenté. Le modèle de données est explicité, les invariants BZP sont donnés sous la forme de formules et les opérations sont exprimées à l'aide de graphes BGP lorsqu'ils présentent des points de choix, ou sous forme de formules sinon.

On notera que les noms des éléments issus du modèle Java/JML, sont, par convention, préfixés par `b_` pour produire un atome Prolog. Ceci nous permet de considérer sans ambiguïté les éléments venant du modèle Java/JML et les éléments ajoutés pour la représentation BZP (par exemple : `modele`, `instances`, etc).

Avant de nous intéresser à l'expression des attributs des classes, prenons un instant pour décrire les types de données et leur représentation en BZP.

5.2.3 Définition des typages des données

Les types supportés par notre approche sont les entiers et les types objets. Le tableau 5.1 récapitule la traduction des types Java en BZP, précisant pour chaque type de donnée le type abstrait et le type concret BZP sous la forme du domaine de valeurs.

Type Java	Type abstrait BZP	Type concret (domaine)
Types basiques :		
byte	int	-128 .. 127
short	int	-32768 .. 32767
int	int	-2^{23} .. $2^{23}-1$
char	int	0..256
boolean	atom	{`true`,`false`}
Types structurés :		
Objet de type statique C	atom	$I_C \cup$ {`null`}

TAB. 5.1 – Expression des types Java en BZP

Nous considérons différentes échelles de valeurs correspondant aux domaines de définition des attributs. Ainsi les différentes variétés d'entiers existant en Java sont supportés. Néanmoins, les entiers du genre `int` sont restreints à un codage sur 24 bits (au lieu de

32 bits en Java). Cette limitation est due au solveur CLP(FD) de SICStus assurant la manipulation des entiers. Par extension, les entiers longs (codés sur 64 bits) ne sont pas supportés. Les valeurs booléennes sont traduites comme des atomes représentant les deux valeurs possibles : `true` pour *vrai* et `false` pour *faux*.

Après avoir défini les types de données et leur représentation en BZP, nous nous intéressons à présent au second élément qui décrit l'état d'un programme Java : les attributs et leurs valeurs.

5.2.4 Représentation des attributs de classe

Les attributs de classe décrivent les propriétés relatives à un objet. Nous présentons ici la modélisation choisie pour représenter les attributs, leurs valeurs initiales, et la notion de visibilité et d'accès aux attributs d'une classe.

Traduction des attributs de classe

Pour chacun des attributs de classe, une variable est déclarée. On distingue alors deux cas de figure : soit l'attribut est statique, soit il s'agit d'un attribut d'instance.

DÉFINITION 6 (ATTRIBUT STATIQUE ET ATTRIBUT D'INSTANCE) *Les attributs déclarés comme* statiques *(avec le mot-clé* `static`*) sont partagés par toutes les instances, par opposition aux attributs* d'instance *dont la valeur est relative à une instance précise.*

Deux cas se présentent alors pour la traduction. (*i*) Les attributs statiques sont exprimés par des variables dont le type est la traduction du type de l'attribut conformément au tableau 5.1. (*ii*) Les attributs d'instance sont traduits par des fonctions totales associant aux instances de la classe la valeur de l'attribut. On utilise des prédicats BZP de type `invariant` pour préciser les domaines des attributs de classe. A l'initialisation, seuls les attributs statiques prennent les valeurs par défaut. Les attributs d'instances, représentés par des ensembles de couples, sont initialisés à l'ensemble vide.

Valeurs initiales

A la création d'une instance de classe, si des valeurs initiales ne sont pas données pour les attributs, alors des valeurs par défaut prennent le relais. La valeur par défaut pour les objets est `null`, pour les entiers cette valeur est 0, et pour les booléens cette valeur est `false`.

Exemple 5.2 (Expression des attributs de classe) Considérons la classe C suivante, contenant deux attributs d'instance de type C et `short`, et un attribut `static` de type `byte`, comme illustré ci-dessous :

```
class C {
    C att1;   static byte att2 = 42;   short att3;
}
```

Les attributs de cette classe sont exprimés en BZP par le modèle de données et les contraintes suivantes :

- Une variable nommée de type `set(pair(atom,atom))` définissant une fonction totale est créée pour représenter l'attribut `att1`.
- Une variable de type `int` est créée pour représenter l'attribut statique `att2`.
- Une variable de type `set(pair(atom,int))` définissant une fonction totale est créée pour représenter l'attribut `att3`.
- L'invariant BZP donne les domaines de ces trois variables :

$$\texttt{att1} \in I_C \rightarrow I_C \cup \{null\} \ \wedge \ \texttt{att2} \in -128..127 \ \wedge \ \texttt{att3} \in I_C \rightarrow -32768..32767$$

  où la notation $X \rightarrow Y$ désigne une fonction totale associant à chaque élément de l'ensemble X un des éléments de l'ensemble Y.
- Initialement, nous avons :

$$\texttt{att1} = \emptyset \ \wedge \ \texttt{att2} = 42 \ \wedge \ \texttt{att3} = \emptyset$$

Comme nous pouvons le constater, `att2` est initialisé à 42 car c'est un attribut `static`. A l'inverse, `att1` et `att3` sont initialisés à l'ensemble vide, car il n'existe initialement pas d'instances dans le système.

Dans notre approche, nous effectuons l'accès aux attributs à travers des accesseurs pour représenter l'accès aux attributs déclarés comme publics.

Accès aux attributs publics

Pour permettre l'accessibilité des attributs de visibilité `public`, nous associons à chaque attribut $att1$ une opération nommée `get_`$att1$ qui permet de récupérer une référence à l'attribut et de le rendre disponible. L'opération admet un paramètre d'entrée nommée *this*, et permet la récupération d'un attribut d'instance $att1$ public d'une classe C. Cette opération est décrite par le comportement suivant :

$$this \in I_C \ \wedge \ this \in A \ \wedge \ A' = A \cup \{C.att1(this)\} \tag{5.12}$$

où A est l'ensemble des instances disponibles défini par la définition 4.

Pour un attribut $att2$ déclaré comme `static`, l'opération ne considère pas d'instance associée comme paramètre, et se résume au comportement suivant :

$$A' = A \cup \{att2\} \tag{5.13}$$

Modification des attributs publics

De manière similaire, nous associons aux attributs `public` une opération permettant d'aller affecter directement une valeur à l'attribut considéré. Cette opération se nomme `set_`$att1$ où $att1$ désigne le nom de l'attribut. L'opération permettant d'affecter une valeur *val* de type *type* à un attribut d'instance $att1$ `public` déclaré dans une classe C est décrite par le comportement suivant :

$$this \in I_C \ \wedge \ this \in A \ \wedge \ val \in dom(type) \ \wedge \ C.att1'(this) = val$$

où $dom(T)$ donne le type concret (le domaine associé au type Java). Comme on peut le constater l'attribut ne peut être modifié que si l'objet qui lui est associé est lui-même disponible (s'il appartient à l'ensemble A).

Pour un attribut *att*2 de type *type* déclaré comme statique l'opération ne considère pas d'instance associée comme paramètre. Elle est ainsi décrite par le comportement suivant :

$$val \in dom(type) \ \wedge \ att2' = val \tag{5.14}$$

Attributs hérités

Les attributs qui sont hérités ne sont pas re-déclarés dans le module représentant la sous-classe. Seuls les attributs hérités et surchargés sont redéfinis.

Dans ce second cas, les références qui sont faites dans la super-classe à ces attributs concernent le contexte de la super-classe et par conséquent les attributs de la super-classe.

Nous reviendrons sur ce point lors de la traduction des prédicats JML en BZP en partie 6.1, où nous expliquerons comment déterminer dans quel module se situe l'attribut référencé dans un prédicat JML.

Correction de l'initialisation du système

Pour terminer sur la modélisation des états objet en BZP, et par la même occasion, pour récapituler les notions mises en place, nous proposons ici des éléments permettant de prouver la correction de la modélisation choisie.

L'objectif est de donner des éléments permettant d'établir l'équivalence entre l'état initial d'un programme Java, tel qu'il est défini dans les documents de référence [GJS00] et l'état initial représenté dans le modèle BZP.

PROPRIÉTÉ 6 (CORRECTION DE L'INITIALISATION DU SYSTÈME) *Soit $\mathcal{M}i_{Java}$ le modèle de l'état initial d'un programme Java. Soit $\mathcal{M}i_{BZP}$ le modèle de l'état initial du BZP associé au programme Java. Suite à l'initialisation du modèle BZP, le modèle mémoire BZP $\mathcal{M}i_{BZP}$ présente les propriétés suivantes :*

- *Aucune instance n'existe.*
- *Les attributs déclarés comme statiques sont accessibles et initialisés à la valeur initiale, si celle-ci est définie, ou à leur valeur par défaut dans le cas contraire.*
- *Les méthodes déclarées comme statiques peuvent être activées.*
- *Les constructeurs par défaut des classes peuvent être activés.*

Les initialisations d'attributs objets n'étant pas permis, nous constatons que le modèle mémoire BZP $\mathcal{M}i_{BZP}$ et le modèle mémoire Java $\mathcal{M}i_{Java}$ présentent les même propriétés du point de vue des instances, des attributs accessibles, et des constructeurs et méthodes activables.

Nous avons présenté dans cette partie les éléments traitant de l'expression et de la traduction des états Java dans le formalisme BZP, prenant en compte d'abord les instances d'objets puis les attributs. Nous nous intéressons désormais à l'expression des transitions du modèle, définies par les méthodes.

5.3 Représentation des transitions

Nous décrivons dans cette partie la représentation des méthodes. Nous commençons par considérer une méthode seule, puis nous introduisons les éléments de traduction relatifs à la présence d'héritage et de redéfinition de la méthode, induisant du polymorphisme d'héritage et/ou du polymorphisme paramétrique.

5.3.1 Représentation des méthodes

Les méthodes sont exprimées par des opérations BZP. La classe dans laquelle la méthode est déclarée étant exprimée comme un module, ce module contient l'opération considérée. Pour éviter toute ambiguïté, le nom de la méthode est généré à partir de sa signature.

DÉFINITION 7 (MÉTHODE STATIQUE ET MÉTHODE D'INSTANCE) *Les méthodes* d'instances *sont invoquées sur un objet hôte, référencé par l'alias* this. *Elles se différencient des méthodes* statiques *(déclarées avec le mot-clé* `static`*) qui ne nécessitent pas d'objet hôte, et ne manipulent que des attributs statiques de la classe considérée.*

Les méthodes d'instance admettent un paramètre nommé *this* qui correspond à l'instance porteuse de la méthode qui est invoquée. Un prédicat déclaré comme précondition de l'opération donne le typage de l'instance comme appartenant à l'ensemble des instances de la classe considérée. De plus, il est requis que l'instance soit *disponible* pour pouvoir supporter l'invocation d'une méthode. Si la méthode est supposée retourner une valeur –i.e., son type de retour est différent de `void`–, une variable de sortie nommée *result* est déclarée. Un prédicat est ajouté aux postconditions pour préciser le typage concret de cette valeur de retour, suivant les domaines définis dans le tableau 5.1. De plus, si la méthode retourne un objet, celui-ci est ajouté à l'ensemble des objets disponibles.

Les paramètres de la méthode considérée sont traduits comme des paramètres d'entrée de l'opération associée. Pour chacun de ces paramètres, une précondition précise le typage concret du paramètre considéré. De plus, les paramètres objets doivent être disponibles dans le contexte considéré pour garantir la cohérence de l'invocation de la méthode.

Exemple 5.3 (Méthode statique et méthode d'instance) Soit une méthode `void methS(short s)` déclarée comme statique. Cette méthode se traduit par une opération BZP admettant un paramètre s. Le typage de s est donné par la contrainte suivante :

$$s \in -32768..32767$$

Soit une méthode `B methI(B b)` déclarée comme une méthode d'instance dans la classe C. Cette méthode s'exprime par une opération BZP admettant deux paramètres d'entrée : l'objet hôte *this* et le paramètre b, et un paramètre de sortie *result*, sur lesquels portent les contraintes suivantes décrivant le typage des paramètres d'entrée :

$$this \in I_C \ \wedge \ this \in A \ \wedge b \in I_B \cup \{null\} \ \wedge \ b \in A \tag{5.15}$$

où A est l'ensemble des objets disponibles, auquel sont tenus d'appartenir l'objet *this* et le paramètre b. Une partie de la postcondition de cette méthode précise le typage de la valeur de retour et sa disponibilité :

$$result \in I'_B \cup \{null\} \ \wedge \ A' = A \cup \{result\} \tag{5.16}$$

5.3.2 Polymorphisme des méthodes Java et héritage

Les appels des méthodes rencontrés dans une expression Java/JML nous amènent à déterminer quelle méthode devra être activée lors de l'exécution de l'appel. Contrairement aux références aux attributs qui peuvent être déterminées de manière statique, les appels de méthodes sont déterminés à l'exécution, et doivent être résolus de manière dynamique. Cette contrainte est due à l'aspect polymorphe de Java.

DÉFINITION 8 (MÉTHODE POLYMORPHE) *Une méthode m dans une classe C est dite polymorphe si elle présente une ou plusieurs des caractéristiques suivantes :*

- *La méthode m possède le même nom qu'une autre méthode dans une autre classe n'ayant aucune relation d'héritage avec la classe C – on parle de polymorphisme ad hoc ;*
- *La méthode m est une surcharge de la méthode m définie dans une des super-classes de C – on parle de polymorphisme d'héritage ;*
- *La méthode m possède une signature similaire (modulo les héritages entre les types objets des paramètres) – on parle de polymorphisme paramétrique.*

En pratique, nous ne considérons pas le polymorphisme ad hoc, qui est résolu par analyse statique du programme. Nous illustrerons deux cas d'un appel de méthode *polymorphe*. Dans un premier cas, nous allons voir le polymorphisme d'héritage (au niveau de l'objet sur lequel est invoquée la méthode). Dans un second cas, nous allons voir le polymorphisme paramétrique (au niveau des paramètres d'une méthode). Cette description nous permet de déduire des contraintes supplémentaires à appliquer sur les paramètres des opérations BZP représentant les méthodes traduites.

Polymorphisme d'héritage

Soient la hiérarchie de classes et le programme principal suivants.

```
class A {

   public void meth() {
      System.out.println(''A'');
   }
}

class A1 extends A {

}

class A2 extends A1 {

   public void meth() {
      System.out.println(''A2'');
   }
}
```

```
class Test {

   public static void main(String[] args) {
      A a = new A();
      A1 a1 = new A1();
      A2 a2 = new A2();

      test(a);
      test(a1);
      test(a2);
   }

   public static test(A x) {
      x.meth();
   }
}
```

Lors de l'appel à `test(A)`, la méthode `meth(void)` qui est appelée est déterminée au moment de l'exécution par le type dynamique du paramètre `x` :

- Lorsqu'on appelle `test(a)`, la méthode `meth()` qui sera exécutée est celle déclarée dans la classe `A`. Ceci produira comme résultat l'affichage de "`A`" à l'écran.
- Lorsqu'on appelle `test(a1)`, la méthode `meth()` qui sera exécutée est relative à la classe `A1`, qui est en réalité, celle déclarée dans la classe `A` et héritée par la classe `A1`. Ceci produira comme résultat l'affichage de "`A`" à l'écran.
- Lorsqu'on appelle `test(a2)`, la méthode `meth()` qui sera exécutée est celle déclarée dans la classe `A2`. Ceci produira comme résultat l'affichage de "`A2`" à l'écran.

Nous remarquons donc que la méthode appelée est déterminée lors de l'exécution en fonction du type dynamique par rapport au type statique attendu déclaré sur l'objet hôte.

Cette constatation nous amène à renforcer l'expression du typage de l'objet sur lequel est invoqué la méthode. Dans le cadre de l'exemple, on ajoute la contrainte BZP suivante, sur le typage de l'objet hôte *this*, dans la description de `meth()` de la classe `A` :

$$this \in I_A \wedge this \notin I_{A2} \tag{5.17}$$

Polymorphisme paramétrique

La notion de polymorphisme au niveau des paramètres exprime le fait que plusieurs méthodes ayant des signatures différentes puissent être invoquées avec le même appel.

Conservons la hiérarchie de classe précédemment définie et considérons la nouvelle classe de test suivante :

```
class Test {

   public static void main(String[] args) {
      A a = new A();
      A1 a1 = new A1();
      A2 a2 = new A2();

      test(a);
      test(a1);
      test(a2);
   }

   public static test(A x) {
      test2(x);
   }

   public static test2(A x) {
      System.out.println(''x est de type A'');
   }

   public static test2(A2 x) {
      System.out.println(''x est de type A2'');
   }
}
```

Dans cette classe de test, la méthode `test2(A)` et `test2(A2)` possèdent des signatures dynamiques différentes, mais possèdent la même signature polymorphe puisqu'elle peuvent toutes deux satisfaire une invocation avec un paramètre de type dynamique `A2`. Néanmoins, le langage Java étant déterministe, il n'existe qu'une seule invocation possible dans un tel cas de figure.

L'appel à `test(A)` réalise un appel sur différentes méthodes `test2` possibles en fonction du type dynamique du paramètre `x` de `test(A)` :

- Si `x` est de type dynamique `A`, c'est la méthode `test2(A)` qui sera invoquée.
- Si `x` est de type dynamique `A1`, c'est également la méthode `test2(A)` qui sera invoquée.
- Si `x` est de type dynamique `A2`, c'est toujours la méthode `test(A)` qui sera invoquée.

Le polymorphisme paramétrique se résoud donc par analyse statique. Celle-ci permet de déterminer quelle méthode est appelée par rapport à sa signature. Le choix de la méthode se fait donc par rapport à celle qui a la signature la plus générale.

5.3.3 Représentation du polymorphisme des méthodes

De la même manière que pour les attributs, les méthodes qui sont héritées ne sont pas re-déclarées dans le module représentant la sous-classe. Lorsqu'une méthode d'instance est surchargée dans une sous-classe, on ajoute une contrainte supplémentaire sur l'appel de la méthode dans la super-classe, pour signifier que la méthode surchargée de la super-classe ne peut pas être invoquée sur une instance de la sous-classe.

Exemple 5.4 (Polymorphisme d'héritage) Considérons la classe *SuperC* dont la classe *C* hérite et dont elle redéfinit la méthode `meth(void)`.

```
class SuperC {

    void meth() { ... }
}
```

```
class C extends SuperC {

    void meth() { ... }
}
```

Les contraintes de typage de l'objet hôte associé à cet héritage sont les suivantes. Tout d'abord, pour la méthode `meth` de la classe *SuperC* :

$$this \in I_{SuperC} \ \wedge \ this \in A \ \wedge \ this \notin I_C$$

Puis pour celle de la classe C :

$$this \in I_C \ \wedge \ this \in A$$

Le polymorphisme paramétrique est également à prendre en compte. Ces méthodes admettent des signatures différentes mais peuvent être invoquées sur un même paramètre. Contrairement au polymorphisme d'héritage, qui dépend du type dynamique de l'objet hôte sur lequel est invoquée la méthode, le polymorphisme paramétrique se résoud de manière statique. Il n'y a donc aucun renforcement de préconditions.

Nous illustrons ce principe par l'exemple ci-après.

Exemple 5.5 (Polymorphisme paramétrique) Considérons la classe C qui définit deux méthodes `meth(B)` et `meth(B1)`, $B1$ héritant de B.

```
class C {

    void meth(B b) { ... }

    void meth(B1 b) { ... }
}
```

```
class B { ... }

class B1 extends B { ... }
```

Les contraintes de typage de l'objet hôte et aux paramètres b associés à ces méthodes polymorphes sont les suivantes. Tout d'abord, pour la méthode `meth(B)` :

$$this \in I_C \ \wedge \ this \in A \ \wedge \ b \in I_B \cup \{null\} \ \wedge \ b \in A$$

Puis pour la méthode `meth(B1)` :

$$this \in I_C \ \wedge \ this \in A \ \wedge \ b \in I_{B1} \cup \{null\} \ \wedge b \in A$$

Pour en terminer avec la représentation d'une classe Java et illustrer le fonctionnement des méthodes, nous décrivons à présent le mécanisme du constructeur de classe par défaut.

5.4 Constructeur d'objet par défaut

La sémantique de Java considère que tous les objets possèdent un constructeur par défaut qui est en charge : (*i*) de réserver l'adresse de l'objet dans le tas, (*ii*) d'affecter les valeurs par défaut aux attributs de l'objet.

Au cas où la classe Java possède également un constructeur défini, un appel implicite au constructeur par défaut est réalisé. Même si nous ne nous intéressons pas encore à l'expression et à l'interprétation des spécifications de méthodes, il est impératif de recréer ce mécanisme pour permettre l'animation des modèles JML.

5.4.1 L'opération BZP `constructor`

Nous considérons systématiquement une opération nommée `constructor` pour chaque classe d'objet. Cette opération est en charge de choisir une adresse non affectée dans le tas et de l'ajouter aux ensembles d'instances des objets déjà existantes. De plus, pour chaque attribut de chaque classe, les valeurs par défaut sont ajoutées aux objets. Pour finir, cette opération retourne la valeur de la nouvelle adresse calculée.

Exemple 5.6 (Constructeur par défaut) Considérons la classe *SuperC*, contenant deux attributs `att1` et `att2`. La classe C hérite de *SuperC*, contient l'attribut `att2` dont il surcharge l'attribut de la super-classe, et l'attribut `att3`. Le code Java correspondant est le suivant :

```
class SuperC {

    SuperC att1;

    byte att2;
    ...
}
```

```
class C extends SuperC {

    byte att2 = 10;

    short att3;
}
```

Le comportement du constructeur de la classe C par défaut exprimé en BZP est donné en figure 5.1, que l'on commente. Tout d'abord la précondition (prédicat 5.18) requiert qu'il existe au moins une adresse qui n'a pas encore été instanciée venant du tas H. Cette vérification est nécessaire car nous travaillons dans des ensembles de données finis et donc la création des objets est limitée à la taille du tas, fixée lors de la création du BZP. L'instance en cours de création, nommée *this*, est tout d'abord instanciée (prédicat 5.19). Elle est ensuite ajoutée aux ensembles d'instances du modèle (prédicat 5.20), de la classe C (prédicat 5.21) et de la classe *SuperC* (prédicat 5.22). Le type dynamique de l'instance est ensuite explicité (prédicat 5.23).

Puis, les attributs de la classe C sont initialisés (prédicats 5.24 et 5.25). Ensuite, les attributs de la super-classe *SuperC* sont initialisés (prédicats 5.26 et 5.27). Dans ces quatre derniers prédicats le traitement effectué est de rajouter aux ensembles de couples représentant les attributs d'instance, le couple associant la nouvelle instance à sa valeur par défaut. Pour finir, l'instance créée est ajoutée à l'ensemble des instances disponibles.

Comme nous le constatons, le mécanisme de construction de l'objet ne fournit pas de valeur concrète, i.e., une adresse mémoire, à l'objet. Il ne lui fournit qu'une valeur contrainte représentant une adresse qui n'est pas encore affectée. L'instanciation des ensembles d'adresses sera expliquée dans la sous-partie 6.4.2 traitant de l'exécution des méthodes JML. Pour l'instant, considérons que cette nouvelle instance possède une adresse, instanciée par un mécanisme de plus haut niveau, externe à la description de l'opération.

$$H \setminus I \neq \emptyset \quad \wedge \tag{5.18}$$
$$this \in H \setminus I \quad \wedge \tag{5.19}$$
$$I' = I \cup \{this\} \quad \wedge \tag{5.20}$$
$$I'_C = I_C \cup \{this\} \quad \wedge \tag{5.21}$$
$$I'_{SuperC} = I_{SuperC} \cup \{this\} \quad \wedge \tag{5.22}$$
$$T' = T \cup \{this \mapsto C\} \quad \wedge \tag{5.23}$$
$$C.att2' = C.att2 \cup \{this \mapsto 10\} \quad \wedge \tag{5.24}$$
$$C.att3' = C.att3 \cup \{this \mapsto 0\} \quad \wedge \tag{5.25}$$
$$SuperC.att1' = SuperC.att1 \cup \{this \mapsto null\} \quad \wedge \tag{5.26}$$
$$SuperC.att2' = SuperC.att2 \cup \{this \mapsto 0\} \quad \wedge \tag{5.27}$$
$$A' = A \cup \{this\} \tag{5.28}$$

FIG. 5.1 – Comportement issu de l'opération `constructor`

5.4.2 Eléments de correction du constructeur par défaut

Cette sous-partie est consacrée à donner des éléments de correction de la modélisation du constructeur d'objet par défaut des classes.

Le constructeur associé en Java est le constructeur par défaut lorsque celui-ci est applicable, c'est-à-dire lorsqu'il n'existe aucun autre constructeur.

La procédure d'exécution du constructeur de classe par défaut en Java est la suivante [GJS00] :

1. Allocation de l'objet en mémoire ;
2. Initialisation des attributs dans les super-classes (suivant cette même procédure) ;
3. Exécution des initialisations statiques de la classe ;
4. Exécution des initialisations des variables de la classe.

Les propriétés suivantes sont établies par l'exécution de l'opération BZP `constructor` simulant l'exécution du constructeur par défaut.

PROPRIÉTÉ 7 (CORRECTION DU CONSTRUCTEUR DE CLASSE PAR DÉFAUT) *Le constructeur de classe par défaut BZP réalise :*
- *la création d'une instance, en allouant une adresse mémoire non affectée à l'objet en cours de création ;*
- *l'initialisation des attributs d'instance à leur valeur initiale, si celle-ci est définie, ou à leur valeur par défaut ;*
- *les attributs associés à chaque super-classe de l'objet considéré sont initialisés de la même manière.*

Le fait que nous n'autorisons pas les initialisations des attributs objets au moment des déclarations garantit ainsi la préservation de la sémantique Java/JML.

L'effet de l'activation de l'opération BZP `constructor`, représentant le constructeur d'objet par défaut de Java permet d'obtenir un état équivalent à l'état Java en terme d'objets existants et de valeurs d'attributs pour ces objets.

5.5 Synthèse

Nous avons présenté dans ce chapitre la modélisation choisie pour représenter la structure des classes Java, à savoir les instances d'objets, les attributs de ces objets et les méthodes. Nous avons ainsi décrit les mécanismes objets dans une notation ensembliste. Ceci nous a permis de mettre en évidence une traduction qui préserve les propriétés initiales d'un système, et qui garantit en plus les contraintes d'intégrité entre les objets par rapport aux relations d'héritage qui peuvent exister entre eux. De plus, nous avons modélisé le fonctionnement de la machine virtuelle Java (JVM) notamment vis-à-vis de la résolution des appels de méthodes polymorphes.

Ce travail a été publié dans l'article [BDG05] où nous avons décrit l'expression de classes Java dans le formalisme des machines abstraites B.

La représentation présentée dans ce chapitre est la base de la traduction des modèles Java/JML dans le format BZP, décrivant le modèle de données Java. Le chapitre suivant s'intéresse à l'expression des modèles JML et à leur interprétation par des systèmes de contraintes en vue de réaliser l'animation symbolique du modèle.

Chapitre 6

Interprétation des spécifications JML

Sommaire

Ce chapitre traite de l'expression des modèles JML sous la forme de systèmes de contraintes. Les résultats présentés dans ce chapitre nous permettent de réaliser l'animation symbolique du modèle qui nous sera utile lors du processus de génération de tests. Cette représentation des modèles JML utilise les possibilités offertes par le solveur de contraintes CLPS-BZ.

Avant d'aborder la traduction des modèles à travers les différentes clauses JML, nous avons besoin de décrire la traduction des prédicats. Nous commencerons donc ce chapitre par l'expression des prédicats JML. Parallèlement aux prédicats, nous nous intéressons à la traduction et à l'interprétation des appels de méthodes contenus dans les prédicats JML.

Nous nous attardons ensuite sur l'intégration des clauses de spécification JML dans le format BZP. Ceci nous permettra d'exprimer les propriétés statiques du système, mais aussi, et surtout, les propriétés dynamiques du système qui nous servent à réaliser l'animation du modèle. Pour terminer, nous décrivons les impacts, liés à la notation orientée-objet, sur le module réalisant l'animation symbolique.

6.1 Traduction des prédicats JML en BZP

Nous traitons tout d'abord l'expression des prédicats JML dans le format BZP. Cette traduction considère l'expression des attributs, puis des appels de méthodes, avant de s'intéresser à l'expression en BZP des opérateurs relatifs aux objets.

6.1.1 Références aux attributs

Les références aux attributs se font à travers l'évaluation de la fonction représentant l'attribut considéré pour les attributs d'instance, ou en accédant directement à la valeur pour les attributs statiques. La difficulté de la référence aux attributs vient du fait qu'en JML les prédicats avant-après distinguent les valeurs avant (par un `\old`) tandis que le format BZP distingue les valeurs après (par un `prime`).

Ainsi, dans les postconditions, les attributs qui ne sont pas sous la portée d'un `\old` sont considérés comme évalués à l'état d'après.

L'expression de la référence à un attribut en BZP est réalisée de la manière suivante. Pour chaque attribut référencé, nous connaissons, par analyse statique, le type statique attendu de l'attribut. Celui-ci fournit le nom de la classe dans laquelle se trouve l'attribut référencé, c'est-à-dire, soit la classe correspondant au type statique de l'attribut, soit la classe dont est hérité l'attribut.

Toute référence à un attribut x déclaré comme `static` dans une classe C se traduit de la manière suivante :

$$[\![x]\!]_{BZP} = C.x \tag{6.1}$$

où la notation $[\![\_]\!]_{BZP}$ désigne la traduction en BZP d'une expression.

Toute référence à un attribut d'instance nécessite d'évaluer l'instance à laquelle on fait référence. Ainsi, la référence à l'attribut $x_1.x_2.\ldots.x_{n-1}.x_n$ accède à l'attribut x_n de l'instance $x_1.x_2.\ldots.x_{n-1}$. Soit C_n la classe associée au type statique de x_n, l'expression $x_1.x_2.\ldots.x_{n-1}.x_n$ se traduit par :

$$[\![x_1.x_2.\ldots.x_{n-1}.x_n]\!]_{BZP} = C_n.x_n([\![x_1.x_2.\ldots.x_{n-1}]\!]_{BZP}) \tag{6.2}$$

La récursion cesse sur la traduction de x_1 selon les deux cas présentés précédemment (attribut d'instance et attribut statique). Implicitement, si x_1 est un attribut d'instance, l'instance associée est *this*.

Exemple 6.1 (Référence à un attribut) Considérons la référence à l'attribut suivant :

```
a.b.c
```

où `a` est un attribut d'instance de type statique `A`. La classe `A` possède un attribut `b` de type statique `B`. La classe `B` possède un attribut `c`, dont le type est `C`. Le code BZP traduisant cette expression est le suivant :

```
C.c(B.b(A.a(this)))
```

où `this` désigne l'instance courante.

6.1.2 Opérateurs arithmétiques et booléens

Les opérateurs arithmétiques équivalents entre Java et le format BZP permettent l'expression aisée des expressions arithmétiques contenues dans les prédicats JML. Les opérateurs booléens de JML trouvent eux aussi leur équivalent direct en BZP. Le tableau 6.1 donne la traduction de ces opérateurs.

Opérateur Java/JML	Notation mathématique	Opérateur BZP
`P1 && P2`	$P_1 \wedge P_2$	$[\![$`P1`$]\!]_{BZP}$ `&` $[\![$`P2`$]\!]_{BZP}$
`P1 || P2`	$P_1 \vee P_2$	$[\![$`P1`$]\!]_{BZP}$ `or` $[\![$`P2`$]\!]_{BZP}$
`! P`	$\neg P$	`not(`$[\![$`P`$]\!]_{BZP}$`)`
`P1 ==> P2`	$P_1 \Rightarrow P_2$	$[\![$`P1`$]\!]_{BZP}$ `=>` $[\![$`P2`$]\!]_{BZP}$
`P1 <==> P2`	$P_1 \Leftrightarrow P_2$	$[\![$`P1`$]\!]_{BZP}$ `<=>` $[\![$`P2`$]\!]_{BZP}$
`(\forall` T_1 p_1`;` P_1`;` P_2`)`	$\forall p_1.p_1 \in T_1 \wedge P_1 \Rightarrow P_2$	`forall(`$[\![T_1]\!]_{BZP}$`,` `b_`$p_1 \in dom(T_1)$`,` $[\![P_1]\!]_{BZP}$ `=>` $[\![P_2]\!]_{BZP}$`)`
`(\exists` T_1 p_1`;` P_1`;` P_2`)`	$\exists p_1.p_1 \in T_1 \wedge P_1 \wedge P_2$	`exists([b_`p_1`],[`$[\![T_1]\!]_{BZP}$`],` `b_`$p_1 \in dom(T_1)$ `&` $[\![P_1]\!]_{BZP}$ `&` $[\![P_2]\!]_{BZP}$`)`

TAB. 6.1 – Traduction des opérateurs booléens de Java et JML en BZP

6.1.3 Opérateurs sur le type des objets

La gestion des objets et de leur type étant rendue explicite au niveau de la modélisation effectuée dans notre approche, la traduction des opérateurs Java/JML relatifs au type des objets en est facilitée.

Nous rappelons que le type des variables est donné par `\TYPE`. L'opérateur de sous-typage JML `<:` n'étant pas directement exprimable en BZP, nous rajoutons un opérateur de type booléen nommé `sub`, qui prend en paramètre deux expressions `E1` et `E2` et qui vérifie que la classe associée à `E1` hérite de la classe associée à `E2`. Cette vérification est assurée par la propriété ci-dessous.

PROPRIÉTÉ 8 (DÉTECTION DE SOUS-TYPAGE) *Soient C_1 et C_2 les classes respectivement associées aux expressions E_1 et E_2. L'opérateur booléen* `sub` *introduit précédemment se définit de la manière suivante :*

$$E_1 \texttt{ sub } E_2 \equiv \quad C_1 = C_2 \ \vee \ \exists\, C_3.C_1 \sqsubseteq_1 C_3 \wedge \ \ C_3 \texttt{ sub } C_2 \tag{6.3}$$

Expression Java/JML	Traduction en BZP
`E instanceof C`	$[\![\texttt{E}]\!]_{BZP} \in I_C$
`\type(C)`	$[\![\texttt{C}]\!]_{BZP}$
`\typeof(E)`	$T_{[\![\texttt{E}]\!]_{BZP}}$
`E1 <: E2`	$[\![\texttt{E1}]\!]_{BZP}$ `sub` $[\![\texttt{E2}]\!]_{BZP}$

TAB. 6.2 – Traduction des expressions de typage des objets en BZP

L'idée de cet opérateur récursif est de remonter dans la hiérarchie d'héritage en partant de la classe de `E1`, pour chercher à atteindre la classe de `E2`. Pour ce faire, nous considérons que si une classe est un sous-type d'une autre classe, alors soit elle hérite directement de cette dernière –opérateur $\sqsubseteq_1$– soit elle hérite d'un des sous-types de cette dernière. Le prédicat échoue si la récursion remonte au plus haut de la hiérarchie des classes sans pour autant avoir trouvé la classe recherchée.

La tableau 6.2 récapitule la traduction des expressions JML sur le typage des objets en BZP. Dans ce tableau, `E`, `E1` et `E2` désignent des expressions Java/JML de type objet ; `C` désigne un nom de classe.

6.1.4 Créations dynamiques d'objets

Les nouveaux objets créés dynamiquement par l'exécution d'une méthode sont signalés par un prédicat JML `\fresh(o)`. Pour utiliser cette information dans le mécanisme d'animation du JML, nous considérons une variable locale à chaque opération, nommée `new_instances` et déclarée comme un ensemble d'atomes. Cet ensemble est typé dans la précondition de l'opération comme un sous-ensemble des adresses mémoires disponibles. La postcondition établit que le nouvel ensemble d'instances globales résultant est égal aux instances à l'état d'avant augmenté des ensembles créés durant l'exécution.

La précondition suivante est ainsi ajoutée à la précondition de la méthode :

$$\texttt{new\_instances} \subseteq H \setminus I$$

Parallèlement, la postcondition suivante est ajoutée :

$$I' = I \cup \texttt{new\_instances} \bigwedge_{C \in AllC} I'_C \setminus \texttt{new\_instances} = I_C$$

où $AllC$ désigne l'ensemble des classes considérées.

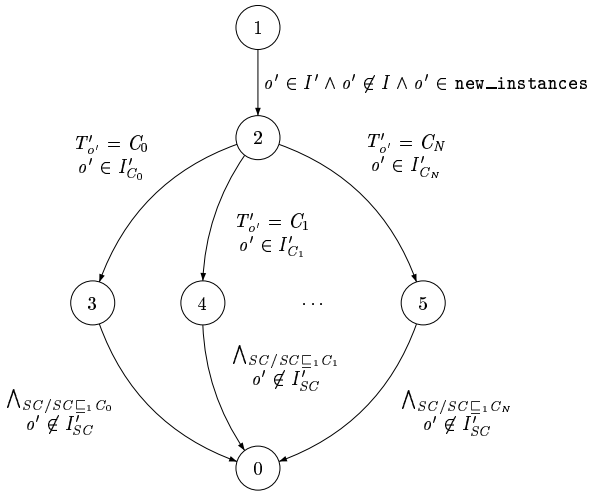
Chaque prédicat `\fresh(x)` est exprimé par une contrainte qui force `x` à être dans l'ensemble des nouvelles instances. Un point de choix est posé autour du type dynamique de `x`, entre le type statique attendu (i.e., déclaré dans le Java/JML et calculé par analyse statique du code/du modèle), et tous les sous-types possibles. Pour chaque type choisi, il faut poser des contraintes d'appartenance de `x` à l'état d'après aux ensembles d'instances correspondant, et de non-appartenance aux ensembles d'instances de chacune des sous-classes. De plus, pour exprimer le type dynamique, il faut fixer la valeur de `typeof` à l'état après.

PROPRIÉTÉ 9 (CHOIX DU TYPE DYNAMIQUE D'UN OBJET) *Soit un attribut objet statique o' de type statique C_0 (dont héritent les classes $C_1, ..., C_N$), et sur lequel porte une postcondition JML* `\fresh(o)`. *Les contraintes BZP suivantes sont associées à cet objet :*

$$o' \in I' \wedge o' \notin I \wedge o' \in \texttt{new\_instances}[]_{C \in C_0, C_1, ..., C_N}(T'_{o'} = C \wedge o' \in I'_C \bigwedge_{SC \sqsubseteq_1 C} o' \notin I'_{SC})$$

où [] est l'opérateur de choix BZP créant un point de choix dans le graphe BGP généré à partir de cette expression.

Le graphe BZP/BGP associé à cet ensemble de contraintes se présente de la façon suivante :



Notons que ce point de choix peut être par la suite déchargé en utilisant un prédicat sur le typage dynamique de o pour en forcer la valeur.

Par la suite, nous considérerons que nous avons toujours à notre disposition des environnements contenant l'état avant du système, l'état après, et les variables locales à la dernière opération qui a été exécutée. Ceci permet d'évaluer une expression `\fresh` dans une contrainte historique, donc non liée à une méthode.

Nous définissons à présent la traduction et l'interprétation des appels de méthodes à l'intérieur des prédicats JML.

6.2 Traduction des appels de méthodes

Les appels de méthodes sont exprimés par des primitives BZP `call`. On distingue alors deux mécanismes pour interpréter les appels de méthode. Le premier, inhérent à Java, consiste à déterminer quelle opération appeler lorsque la méthode est potentiellement polymorphe. Le second mécanisme, relatif à JML, consiste à donner la possibilité d'interpréter les méthodes déclarées comme `pure` aussi bien dans le contexte de l'état avant ou dans le contexte de l'état après.

6.2.1 Traduction d'appels de méthodes polymorphes

Dans le cas d'une méthode polymorphe, un point de choix est posé entre les différentes opérations, représentant la méthode cible, qu'il est possible d'appeler à partir du type statique considéré. On associe à l'appel de la méthode polymorphe un prédicat représentant la condition d'appel de la méthode désignée. Illustrons ceci sur un exemple.

Exemple 6.2 (Appel de méthode polymorphe) Soit la hiérarchie de classes suivante :

```
class C {

    void meth(A a) { ... }
}

class C1 extends C {

}
```

```
class C2 extends C1 {

    void meth(A a) { ... }

    void meth(A1 a) { ... }
}

class A { ... }

class A1 extends A { ... }
```

Considérons l'appel de méthode `c.meth(a)` dans lequel `c` est un paramètre de type statique C, ce qui signifie que le type dynamique de `c` est soit C, soit $C1$, soit $C2$ (ou toute autre sous classe), et `a` est un paramètre de type statique A, ce qui signifie que son type dynamique est soit A, soit $A1$ (ou tout autre sous classe).

La traduction de `c.meth(a)` en instruction BZP sera réalisée par la pose d'un point de choix entre :

- L'appel à la méthode `meth(A)` de la classe C si `c` est de type dynamique C ou $C1$.
- L'appel à la méthode `meth(A)` de la classe $C2$ si `c` est de type dynamique $C2$ et quelque soit le type dynamique de `a` (A, $A1$ ou tout autre sous classe).

Les conditions d'activation guidant ces choix sont connues grâce à l'analyse statique du modèle Java/JML. Celle-ci nous permet également de déduire que la méthode `meth(A)` relative à la classe $C1$ est en réalité héritée de la classe C.

La primitive `call_fd` est une sur-couche à l'appel de la primitive `call` standard. L'appel de méthode sur notre exemple est ainsi traduite en BZP par la primitive `call_fd` suivante :

$$\texttt{call\_fd}(C\texttt{.meth}(A)\texttt{, [c, a]})$$

L'algorithme d'activation de l'appel à l'opération `meth` est donné en figure 6.1.

Cet algorithme consiste à tenter l'activation de l'opération du module attendu. Si celle-ci échoue, ce qui est dû soit (i) à l'absence de l'opération dans le module, soit (ii) à l'insatisfaisabilité de la précondition sur le paramètre représentant l'objet *this*, l'algorithme va tenter d'activer l'opération dans chacun des modules représentant chacune des sous-classes directes.

Ce calcul termine par l'activation de la méthode considérée.

PROPRIÉTÉ 10 (CORRECTION DE LA TRADUCTION DE L'APPEL DE MÉTHODE) *La correction de l'appel de méthode vise à s'assurer que la méthode qui sera exécutée en BZP*

```
resultat ← call_fd(Module.Operation, Params) =
  si (Module, Operation) existe alors
    si resultat = call(Module.Operation, Params) réussit alors
      retourner(resultat)
    finsi
  sinon
    pour tous Module1 ⊑1 Module faire
      si resultat = call_fd(Module1.Operation, Params) réussit alors
        retourner(resultat)
      finsi
    fin pour
  finsi
  échouer
```

FIG. 6.1 – Algorithme de la primitive `call_fd`

est bien la méthode qui sera exécutée, à partir des mêmes conditions en Java. Nous nous basons sur les éléments suivants.

- *L'analyse statique et le contrôle de type nous garantissent qu'il existe au moins une méthode qui est référencée par l'appel. Cette méthode sera donc traduite en BZP.*
- *La traduction du typage de l'objet hôte dans les préconditions ne fournit que des cas mutuellement exclusifs, ceci garantit qu'il n'y a qu'un seul choix parmi les opérations représentant une méthode polymorphe dans les différents modules.*
- *L'appel initial se base sur le type statique de l'objet hôte au moment de l'appel de la méthode. Ceci correspond au niveau le plus haut dans la hiérarchie des classes vérifiant le contraintes de typage de l'objet hôte.*
- *Si l'opération ne s'active pas dans le module considéré, on cherche dans les sous-modules, une opération similaire, en répétant le même mécanisme. Étant donné que l'ensemble de classes est fini, que la hiérarchie de classe est bornée dans la profondeur, et qu'il n'existe pas de cycles dans les relations d'héritages de classes, la récursion terminera.*

A partir de tous ces éléments, nous pouvons conclure que l'appel de méthode terminera par l'activation de la méthode correcte, tant sur le point de la localisation du module de l'opération considérée, que sur le point du choix de l'opération dans un même module. On note au passage que la sémantique originelle du BZP est respectée : si l'appel à la méthode échoue c'est que ses comportements sont inconsistants.

Intéressons-nous maintenant aux appels de méthodes `pure` contenues dans les prédicats JML.

6.2.2 Traduction d'appels de méthodes `pure`

Les méthodes `pure` sont les seules à pouvoir être invoquées dans le cadre d'un prédicat. Notre objectif est donc de permettre l'invocation de ces méthodes.

Disponibilité de l'objet hôte Les objets sur lesquels sont invoquées les méthodes sont supposés être disponibles. Néanmoins, à l'intérieur d'un prédicat, il est possible qu'une méthode soit invoquée sur un objet qui n'est pas disponible de l'extérieur de la méthode. Pour pallier ce problème, les méthodes `pure`, qui sont les seules à pouvoir être invoquées dans un prédicat de spécification JML sont déchargées de cette contrainte. Il en va de même pour les paramètres objets de ces méthodes. Ceci sera illustré dans l'exemple 6.3.

Appels avant ou après Les appels de méthodes `pure` peuvent avoir lieu soit à l'état avant, soit à l'état après. De ce fait, les méthodes ne travaillent pas sur les même attributs si elles sont appelées dans le contexte de l'état avant (dans la précondition ou dans une postcondition sous la portée d'un `\old`) ou à l'état après (dans une postcondition). Ainsi, les méthodes déclarées comme `pure` s'expriment en BZP par deux opérations, l'une pour les appels relatifs à l'état avant, et l'autre relative aux appels à l'état après. Dans le premier cas, les attributs (même dans les postconditions) sont évalués à l'état avant, tandis que dans le second cas, les attributs sont systématiquement évalués à l'état après. C'est l'analyse statique du prédicat qui permet de déterminer quelle opération sera appelée.

Exemple 6.3 (Appel de méthode avant et après) Soit la méthode `pure getVal()`, de la classe C, qui retourne la valeur de l'attribut de classe `val` de type entier court (`short`). Cette méthode peut être appelée dans un prédicat avant-après, comme illustré ci-dessous :

```
//@ constraint  this.getVal() >= \old(this.getVal())
```

Nous considérons donc deux opérations distinctes, respectivement pour l'état avant et l'état après, nommées `getVal` et `getVal'`. Le comportement de l'opération `getVal` du module C est donné par :

$$this \in I_C \ \wedge \ result \in -32768..32767 \ \wedge \ result = C.val(this)$$

et le comportement de l'opération `getVal'` du module C est donné par :

$$this \in I_C \ \wedge \ result \in -32768..32767 \ \wedge \ result = C.val'(this)$$

Comme nous pouvons le constater, alors que la première opération devrait considérer des attributs de la postcondition à l'état d'après, ceux-ci sont considérés à l'état avant pour répondre à un appel de cette méthode dans une précondition, ou sous la portée d'un `\old`. A l'inverse, la seconde méthode vise à répondre à une invocation à l'état après et les attributs référencés dans cette méthode ne sont considérés qu'à l'état après. Ainsi la contrainte historique précédente se traduira par :

$$\texttt{call\_fd}(C.\texttt{getVal'},[this]) \texttt{ >= call\_fd}(C.\texttt{getVal},[this])$$

Au passage, nous remarquons que la contrainte de disponibilité de l'objet hôte, habituellement exprimée par une précondition $this \in A$ n'est pas exprimée dans ces opérations, pour les raisons évoquées précédemment.

6.3 Expression des spécifications JML en BZP

Nous décrivons dans cette partie l'expression des modèles JML dans le format BZP. Nous commençons par présenter l'expression des spécifications de types, avant de nous concentrer sur la traduction des spécifications de méthodes.

6.3.1 Clauses de spécification de type JML

Nous exprimons les clauses de spécification de type JML, à savoir les clauses `initially`, `invariant` et `constraint` dans des prédicats BZP. Néanmoins, pour faire une distinction entre les prédicats BZP habituels et les prédicats BZP issus de la spécification de type, nous créons de nouveaux genres de prédicats, qui viennent s'ajouter aux genres préexistants qui sont `static`, `invariant`, `initialisation`, `pre`, et `post`.

Notre souci principal est de distinguer les propriétés issues du modèle JML des propriétés BZP exprimant les structures des données Java. Comme nous l'avons vu au chapitre 5, la structure du modèle de données Java est exprimée à l'aide de l'invariant BZP qui exprime un état mémoire Java sous la forme d'un ensemble de contraintes.

Pour permettre d'exprimer ces trois types de propriétés JML, nous introduisons trois nouveaux genres de prédicats : `jml_invariant`, `jml_constraint`, `jml_initially`, représentant, respectivement, l'invariant (clause `invariant`), les contraintes historiques (clause `constraints`) et les contraintes initiales (clause `initially`).

Les propriétés JML exprimées dans ces prédicats sont représentés dans le format BZP sous une forme spécifique n'interférant pas avec la représentation du modèle de données ou le processus d'animation. Ainsi ces prédicats extraits du modèle pourront être utilisés pour calculer les cibles de test, comme illustré au chapitre 7, ou vérifiés lors de l'évaluation symbolique d'un modèle, comme illustré dans le chapitre 8.

Exemple 6.4 (Clauses de spécifications de type) Considérons la classe `LimitedPurse` de l'exemple présenté en partie 4.4.2. Les clauses de spécifications de type sont exprimées en BZP de la manière suivante :

```
predicat(LimitedPurse, jml_invariant, 42, Purse.balance(p_this) =< LimitedPurse.max).

predicat(LimitedPurse, jml_initially, 43, LimitedPurse.max = 10000).
```

Parmi ces deux prédicats, le prédicat indexé par le numéro 42 illustre l'invariant `balance <= max`, et celui qui porte le numéro 43 spécifie la contrainte initiale `max == 10000`.

Chacune des clauses étant liée à une classe, et par conséquent à une instance, nous introduisons un objet `p_this` qui représente l'objet sur lequel sont vérifiées les propriétés.

6.3.2 Clauses de spécifications de méthodes

La traduction des clauses de spécifications de méthodes nous permet d'exprimer les transitions entre deux états. Les préconditions restreignent l'ensemble des états avant

possibles et les postconditions restreignent les états après. Nous illustrons l'expression des spécifications de méthodes JML "sans sucre syntaxique" suivant le principe décrit en figure 4.3 dans le chapitre 4.

Préconditions Nous souhaitons permettre l'expression des préconditions JML indépendamment des préconditions BZP déjà utilisées pour le typage des paramètres des méthodes. Suivant la même logique, nous introduisons un nouveau prédicat BZP, nommé `jml_pre(`*Op*`)`, qui exprime la précondition issue du modèle JML, de la méthode représentée par l'opération *Op*.

Divergence De la même manière, nous considérons un nouveau genre de prédicat BZP, nommé `jml_diverges(`*Op*`)` pour exprimer la divergence –éventuelle– d'une méthode, c'est-à-dire, sa non-terminaison. La motivation de cette construction est de conserver les informations sur la divergence d'une méthode. Tout comme pour les clauses de spécification de type, nous utiliserons cette clause à des fins de vérification lors de l'animation du modèle.

Champs modifiés La prise en compte des champs modifiés va jouer un rôle prépondérant dans l'animation. En effet, la connaissance des attributs modifiés, et qui sont donnés dans la clause `assignable`, nous permet de déduire le complémentaire ; les champs ne figurant pas dans cette liste ne changent pas de valeur.

Cette information nous permet d'expliciter la postcondition (initialement implicite) cachée sous la clause `assignable`. L'utilisation de cette clause est liée à la représentation que nous avons choisie pour les attributs d'instances. Celle-ci est une fonction totale qui associe une instance à sa valeur. Une contrainte liée au BZP est ne pas pouvoir modifier une partie seulement de cette fonction totale : soit aucun couple n'est modifié, soit tous les couples sont modifiés. Dans le cas où certains attributs ne sont pas modifiés par l'exécution de la méthode, cela se traduit par le fait que leur valeur après est identique à leur valeur avant.

Pour calculer cette postcondition, nous regroupons les éléments qui représentent le même attribut d'instance. Illustrons ceci sur un exemple, dans lequel nous considérons la classe C suivante, agrémentée d'une méthode `meth(`C`)` :

```
class C {

    T1 att1;
    T2 att2;

    //@ assignable att1, p1.att1, p1.att2;
    public void meth(C p1) { ... }
}
```

La clause `assignable` indique que l'attribut `att1` est modifié pour `this` et `p1`, et que `att2` est modifié pour `p1`. Implicitement, (i) pour toutes les instances différentes de `this` et `p1`, la valeur de `att1` reste identique, et (ii) pour toutes les instances différentes de `p1`, la valeur de `att2` reste inchangée. De plus, les nouvelles instances peuvent prendre n'importe quelle valeur de leur domaine.

Ainsi, les nouvelles valeurs des attributs `assignable` sont explicitées par une contrainte de typage, exprimée en postcondition de l'opération. Sur notre exemple :

$$C.\texttt{att1'(this)} \in dom(T_1) \land C.\texttt{att1'(p1)} \in dom(T_1) \land C.\texttt{att2'(p1)} \in dom(T_2)$$

La clause `assignable` nous permet de considérer que les attributs qui n'ont pas été modifiés restent à la même valeur. Sur notre exemple :

$$\forall\, c.c \in I_C \ \land \ c \neq this \land c \neq \texttt{p1} \Rightarrow C.\texttt{att1'}(c) = C.\texttt{att1}(c)$$

exprime que les attributs `att1` des objets différents de `this` et `p1` restent à la même valeur, et

$$\forall\, c.c \in I_C \land \ \neq \texttt{p1} \Rightarrow C.\texttt{att2'}(c) = C.\texttt{att2}(c)$$

exprime que les attributs `att2` des objets différents de `p1` restent à la même valeur.

En pratique, l'expression des attributs non modifiés est réalisée à l'aide d'une contrainte ensembliste (par souci de performance du moteur d'interprétation) utilisant une soustraction de domaine $\lhd$ qui se définit de la manière suivante :

$$s \lhd t \equiv \{(x, y) \mid (x, y) \in t \land x \notin s\} \tag{6.4}$$

où s est un sous-ensemble du domaine de t.

Ainsi la postcondition issue de la clause `assignable` de l'exemple précédent s'exprime de la manière suivante :

$$\begin{aligned} \texttt{(\{this, p1\} } \cup \texttt{ new\_instances)} \lhd \texttt{att1}' &= \texttt{\{this, p1\}} \lhd \texttt{att1} \\ \texttt{(\{p1\} } \cup \texttt{ new\_instances)} \lhd \texttt{att2}' &= \texttt{\{p1\}} \lhd \texttt{att2} \end{aligned}$$

Cette postcondition prend ainsi en compte les attributs des instances modifiées mais également les attributs des nouveaux objets éventuels, qui ne sont pas précisés dans la clause `assignable`.

Postcondition La postcondition établie dépend de la terminaison considérée. Pour permettre de conserver l'information de la terminaison activée, un paramètre spécifique à l'opération est introduit. Nous avons choisi d'utiliser un paramètre d'entrée pour permettre à un utilisateur de choisir explicitement le comportement qu'il souhaite activer.

Ce paramètre est représenté par T, pour "terminaison". Il peut prendre les valeurs `no_exception` ou E_{ij}. `no_exception` permet d'identifier le comportement entraînant la terminaison normale. E_{ij} identifie une classe d'exception spécifiée dans une clause `signals` de la méthode considérée.

```
/*@ behavior
  @    requires P1;
  @    assignable A;
  @    ensures Q1;
  @    signals (E11 e11) S11;
  @    ...
  @    signals (E1M e1M) S1M;
  @ also
  @    ...
  @ also
  @    requires PN;
  @    assignable A;
  @    ensures QN;
  @    signals (EN1 eN1) SN1;
  @    ...
  @    signals (ENP eNP) SNP;
  @*/
Type meth(T1 p1, ...) throws E1, ..., EN { ... }
```

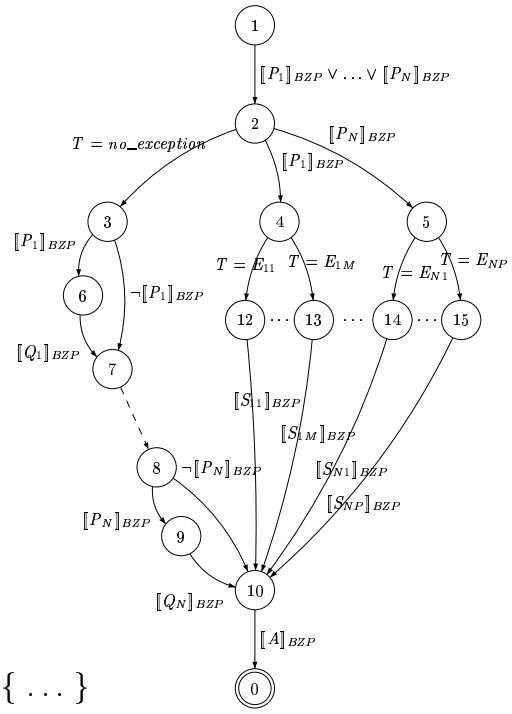


FIG. 6.2 – Expression sous forme d'un graphe BGP d'une spécification de méthode JML

La figure 6.2 illustre la représentation sous forme d'un graphe BGP de la postcondition générée en fonction d'une spécification de méthode JML la plus générale possible. Dans cette figure, la notation $[\![P]\!]_{BZP}$ désigne la traduction du prédicat P dans le format BZP.

La spécification d'une méthode est exprimée dans une et une seule postcondition BZP. Cette postcondition contient les éléments suivants :

- la précondition JML, traduisant les différentes clauses `requires` ;
- un choix entre les comportements possibles :
  (i) le comportement normal, traduisant les différentes clauses `ensures` ;
  (ii) chacun des comportements exceptionnels, traduisant chacune des différentes clauses `signals` ;
- l'information implicite sur les champs non-modifiés.

Nous décrivons à présent comment les spécifications de méthodes JML sont exprimées dans le formalisme BZP.

6.3.3 Expression des méthodes Java annotées en JML

Pour une méthode considérée, dont la signature et les spécifications sont données en figure 6.2, nous donnons ici l'expression en BZP de cette modélisation. Cette traduction s'appuie sur les éléments définis précédemment sur l'expression de chacune des clauses JML dans le formalisme BZP. Les prédicats relatifs au typage de l'objet hôte (si la mé-

thode n'est pas statique) et des paramètres (si la méthode en admet) sont ajoutés au début de graphe donné en figure 6.2.

Le graphe BGP ainsi produit calque le graphe issu de la spécification de méthode JML. Nous proposons des éléments de preuve de la correction de la traduction des spécifications de méthodes JML.

PROPRIÉTÉ 11 (CORRECTION DE LA TRADUCTION D'UNE SPÉCIFICATION DE MÉTHODE)
Considérons un état mémoire $\mathcal{M}$ décliné en une version Java $\mathcal{M}_{Java}$ et son équivalent BZP $\mathcal{M}_{BZP}$. Soient $\mathcal{M}'_{Java}$ et $\mathcal{M}'_{BZP}$ les états mémoires Java et BZP suite à l'exécution de la méthode considérée. Nous faisons l'hypothèse que les états BZP $\mathcal{M}_{BZP}$ et $\mathcal{M}'_{BZP}$ sont correctement structurés du point de vue de la hiérarchie de classes représentées.

Nous considérons l'invocation de la méthode d'instance m sur un objet o, avec les paramètres $p_1, \ldots, p_N$. Cette invocation requiert implicitement :

- *L'existence de l'objet o et la disponibilité de l'objet o par rapport au contexte dans lequel est réalisée l'invocation. Nous considérons que l'invocation d'une méthode sur un objet est effectuée depuis un programme principal. Ainsi, les seuls objets disponibles sont ceux ayant été créés ou ceux ayant été retournés par un appel de méthode.*
- *Le typage correct des paramètres p_i avec $i \in 1..N$. Les paramètres objets doivent exister dans l'environnement et être disponibles.*
- *La création possible de nouveaux objets durant l'exécution de la méthode.*

Les deux premières propriétés sont garanties par la traduction du prototype de la méthode m, données en partie 5.3. La dernière propriété est garantie par l'expression des nouvelles instances données en partie 6.1.4. Ainsi, les états mémoire $\mathcal{M}_{Java}$ et $\mathcal{M}_{BZP}$ possèdent les même propriétés relatives à l'activation de la méthode m considérée.

La spécification JML de la méthode m requiert les éléments suivants (nous nous référons au graphe donné en figure 6.2) :

- *La méthode m n'est activable que si l'une des préconditions P_i avec $1 \leq i \leq N$ est satisfaite. Ceci est exprimé dans le graphe BGP par l'arc entre les nœuds 1 et 2.*
- *La méthode m termine soit en établissant la postcondition normale, soit en établissant une des postconditions exceptionnelles, l'exception associée étant alors déclenchée. Ceci est exprimé dans le graphe BGP par un choix entre les différents comportements à partir du nœud 2 du graphe.*
- *Seules les références données dans la clause* `assignable` *sont autorisées à être modifiées. Les autres références conservent la même valeur. Ceci est exprimé par l'arc entre les nœuds 10 et 0 du graphe, qui s'appuient sur la traduction de la clause* `assignable` *donnée en sous-partie 6.3.2.*
- *Finalement, les instances existantes avant l'exécution existent encore après et seules les nouvelles instances ont été créées, puisqu'on suppose qu'il n'y a pas de mécanisme de* garbage collection.

Ainsi, les états mémoire $\mathcal{M}'_{Java}$ et $\mathcal{M}'_{BZP}$ possèdent les mêmes propriétés suite à l'exécution de la méthode m considérée.

Pour récapituler, nous illustrons la traduction complète d'une des méthodes de notre exemple fil rouge.

Exemple 6.5 (Expression de la spécification de la méthode `debit(short)`) Considérons la méthode `debit(short)` présentée en partie 4.4.2, dont nous donnons ici la signature et la spécification associée, réécrite sous la forme générale.

```
/*@ behavior
  @   requires a > 0 && a <= balance;
  @   assignable balance, hist;
  @   ensures balance == (short)(\old(balance) - a);
  @   ensures \fresh(hist) && hist.getBalance() == \old(balance) &&
  @               hist.getPrevious() == \old(hist);
  @ also
  @   requires a =< 0 || a > balance;
  @   assignable balance, hist;
  @   ensures false;
  @   signals (WrongParameterException) balance == \old(balance) &&
  @                                      hist == \old(hist);
  @*/
public void debit(short a) throws WrongParameterException { ... }
```

Le comportement BZP associé à cette méthode, et sa représentation BGP sont donnés en figures 6.3 et 6.4. Dans ces figures, par souci de concision, la précondition $a > 0 \wedge a \leq Purse.balance(this)$ est exprimée par Pre_1, et la précondition $a \leq 0 \vee a > Purse.balance(this)$ est exprimée par Pre_2. Remarquons au passage que $Pre_1 = \neg Pre_2$, les deux comportements identifiés dans les clauses `ensures` sont donc mutuellement exclusifs.

Commentons cet exemple. Tout d'abord, le domaine de la variable `new_instances` est donné par l'arc 1–2. La méthode `debit(short)` n'étant pas statique, elle admet en paramètre un objet hôte *this* sur lequel elle est invoquée, dont le typage est donné par l'arc 2–3 du graphe. De plus, l'objet hôte doit être disponible (arc 2–3). Le paramètre de la méthode est typé par l'arc 3–4. L'arc 4–5 exprime la précondition JML de la méthode (c'est-à-dire la disjonction des préconditions des blocs de spécification de méthodes). Les arcs entre les nœuds 5 à 13 passant par le nœud 6 donnent les comportements relatifs à la terminaison normale de la méthode. Les arcs entre les nœuds 5 et 13 et passant par le nœud 14 expriment le comportement exceptionnel déclenchant la levée de l'exception de type `WrongParameterException`. Pour finir, les arcs 13–17 et 17–18 expriment les contraintes issues de la clause `assignable` : modification des champs `balance` et `hist` du seul objet *this* (arc 17–18), non-modification du champ `max` de la classe `LimitedPurse` (arc 13–17). Pour finir, la limitation des nouvelles instances aux seules créées lors de l'exécution de la méthode est exprimée par l'arc 18–0.
On remarque dans l'expression de la postcondition de `debit(short)` donnée en figure 6.4, le traitement destiné à convertir un entier X en entier court, en réalisant :

$$((X + 32767)\ mod\ 65535) - 32767$$

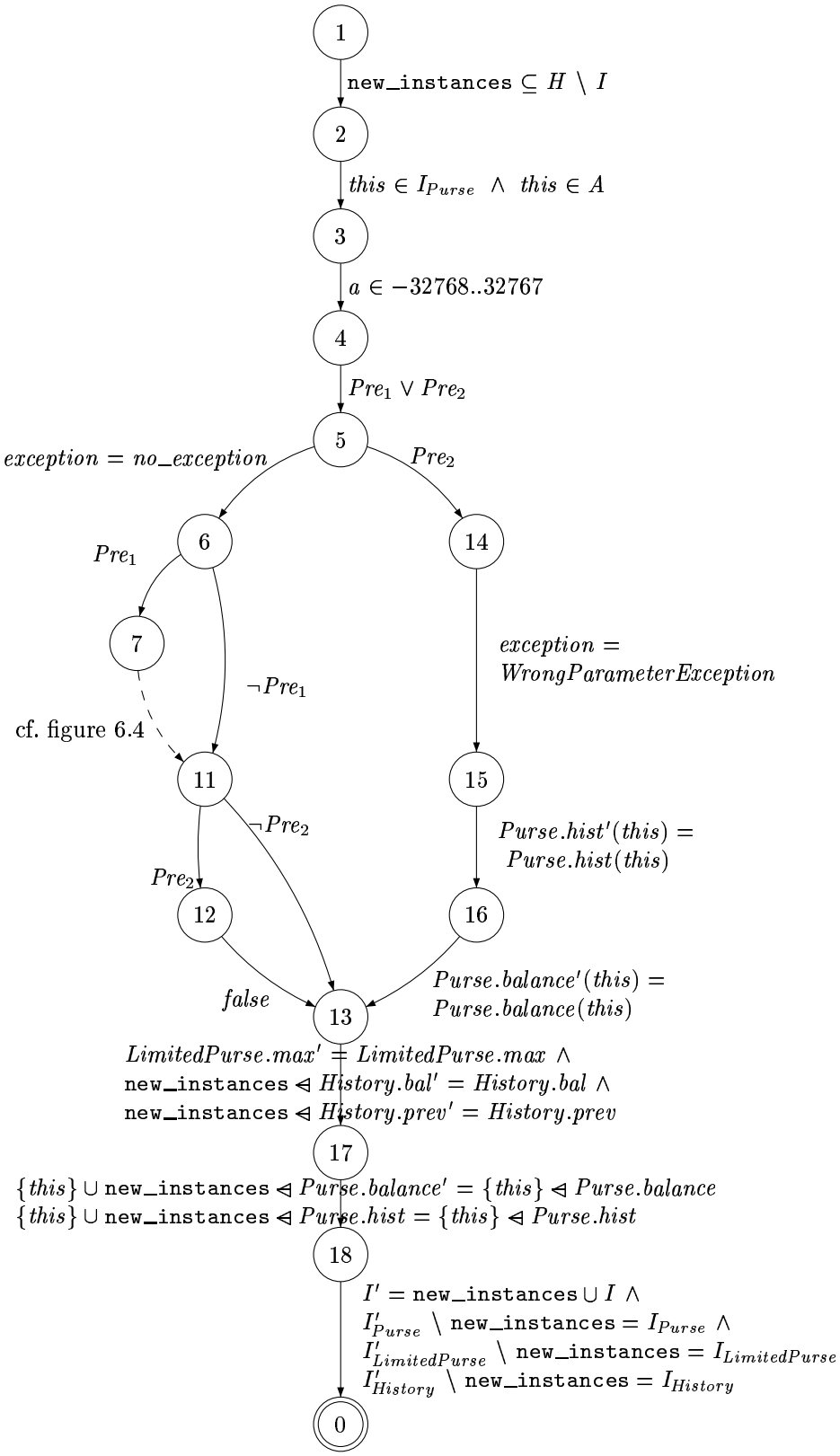


FIG. 6.3 – Graphe BGP de la méthode `debit(short)` de l'exemple

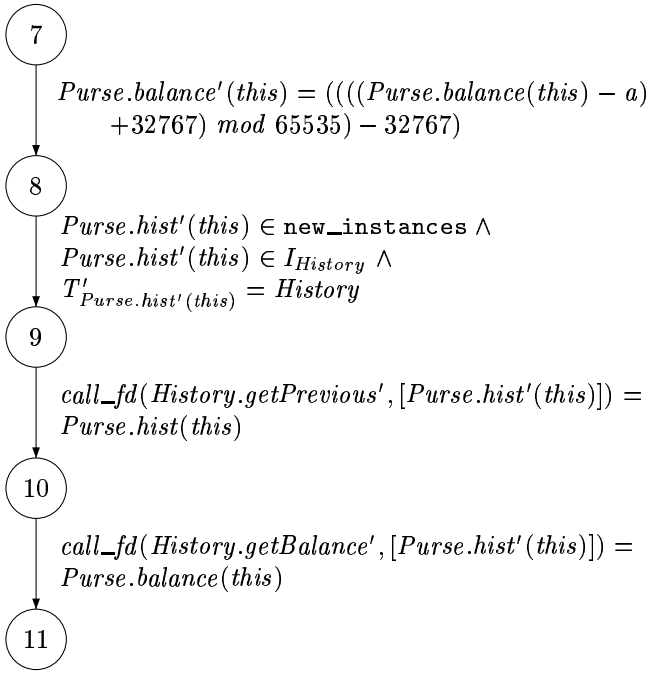


FIG. 6.4 – Graphe de la postcondition normale de la méthode `debit(short)`

Comme nous venons de le voir, une méthode est exprimée par un graphe BGP représentant ses différents comportements. Les constructeurs de classe différents du constructeur par défaut sont exprimés de la même manière qu'une méthode.

Les contraintes relatives aux comportements considérés laissent planer quelques zones d'ombres sur la manière dont est réalisée l'animation symbolique des modèles objets. En effet, l'exécution d'une méthode peut créer *a priori* un nombre inconnu de nouvelles instances, ce qui est exprimé par la variable locale `new_instances` dont le domaine est défini comme un sous-ensemble des adresses libres du tas. Suite à l'exécution de la méthode, dans l'état actuel des choses, cette variable est toujours contrainte car elle n'a pas pu être instanciée durant l'interprétation du comportement issu de la méthode. Or, pour permettre de réaliser une animation qui ait un *sens* pour un utilisateur, il faut donner la possibilité de *fixer* cet ensemble d'instances. Puisque ceci ne peut être réalisé par les contraintes relatives au comportement de la méthode (tant que toutes les contraintes ne sont pas interprétées, il est impossible de connaître toutes les instances qui existent à l'état d'après), il faut mettre en place un mécanisme de plus haut niveau qui résoudra ce problème.

Ceci impacte directement le module d'exécution symbolique, pour lequel nous décrivons et justifions à présent les évolutions.

6.4 Impacts sur l'interprétation à contraintes

L'animation symbolique de modèles JML s'appuie sur l'expression des spécifications Java/JML présentées dans les parties précédentes. Comme nous l'avons vu, la représentation choisie est basée sur la pose de contraintes entre l'état avant et l'état après la transition, pour respecter la sémantique des spécifications de méthodes JML.

Néanmoins, dans l'objectif d'animer le modèle, de manière à calculer des séquences de test, nous sommes obligés de détourner cette sémantique. Nous imposons donc des "choix d'exécution" du modèle, qui impactent les modules d'animation définis dans BZ-TESTING-TOOLS. Ces choix d'implantation et de représentation nécessitent la modification de l'API d'exécution. En effet, l'introduction possible de nouveaux objets dynamiquement créés et l'initialisation en deux temps des attributs de classe par les constructeurs de méthode imposent de faire évoluer le module Executer.

Cette partie s'intéresse donc à décrire les évolutions liées à ce module, pour permettre d'animer les modèles JML à partir des contraintes posées lors du franchissement d'une transition. Nous verrons dans un premier temps les impacts liés à la création dynamique des objets, puis nous nous intéresserons aux impacts liés aux constructeurs de classe. Pour finir, nous donnerons un exemple simple d'animation de la spécification donnée par l'exemple fil rouge en partie 4.4.2, qui permettra de clarifier les apports des modifications effectuées.

6.4.1 Création dynamique d'objets

Lors de l'évaluation des contraintes issues du graphe BGP représentant la spécification d'une méthode JML, l'interpréteur est amené à créer des variables contraintes. C'est en particulier le cas de la variable locale `new_instances`, sous-ensemble de l'ensemble des adresses non encore affectées. Durant l'interprétation des prédicats exprimant l'exécution d'une méthode, seules des contraintes d'appartenance des nouvelles instances sont posées sur cet ensemble. Ceci introduit de nouvelles variables contraintes, une pour chaque nouvelle instance désignée par une expression JML `\fresh`.

Le problème vient du fait qu'il est impossible *a priori* de connaître par exécution les nouvelles instances créées. Il est donc nécessaire, une fois que toutes les contraintes sur l'état d'après ont été posées d'exprimer le fait que l'état après est *clos*, c'est-à-dire qu'il ne contient que les instances qui ont été explicitées : les anciennes instances et celles nouvellement créées.

Puisque nous nous imposons de toujours connaître les valeurs de nos instances et de nos ensembles d'instances, nous effectuons une *instanciation minimale* (qui est une option du solveur CLPS-BZ) de toutes les variables représentant des ensembles d'instances dans l'état après, et de la variable locale `new_instances`.

Pour éviter d'obtenir des ensembles d'instances incohérents, induits par le fait que les seules contraintes posées par les postconditions des opérations peuvent être insuffisantes en particulier pour décrire les valeurs des champs des nouveaux objets, nous posons au

préalable sur *l'état après*, les contraintes exprimant la hiérarchie des classes et le typage des données, représentées par l'invariant BZP.

Ainsi, l'exécution d'une méthode JML se réalise par l'invocation d'un nouvelle primitive, sous la forme $jml\_executeMethod(C, M, P, E)$, qui prend en paramètres la classe C et la méthode M à invoquer avec les paramètres P à partir de l'état décrit par l'environnement E. Cette primitive calcule l'état après l'exécution et l'éventuelle valeur de retour. Elle effectue la succession d'actions suivante :

1. Retrait des valeurs obsolètes de l'environnement E (variables locales à l'opération précédentes, remplacement des valeurs des variables par les valeurs après l'opération précédente) ;
2. Ajout des entrées correspondant aux valeurs des variables d'état après (du genre `prime`) ;
3. Pose des contraintes relatives à la structures des classes à l'état après ;
4. Ajout des paramètres P dans l'environnement d'exécution ;
5. Exécution du graphe BGP correspondant à l'opération M du module C ;
6. Instanciation des ensembles d'instances à l'état d'après pour clore la mémoire ;
7. Récupération de l'éventuelle valeur de retour dans l'environnement.

PROPRIÉTÉ 12 (CORRECTION DE L'EXPRESSION DES NOUVEAUX OBJETS) *Nous cherchons ici à prouver que l'interprétation d'une méthode ne créé pas plus d'objets que nécessaire et que la structure de l'état résultant est cohérent.*

Les nouveaux objets créés sont considérés comme étant en nombre minimal, ce qui est obtenu par l'instanciation de l'ensemble `new_instances`.

La structure des classes est cohérente puisque les contraintes portant sur la hiérarchie des classes et des instances d'objets existants sont posées sur l'état après. De plus, la pose des contraintes de typage des attributs des objets permet d'exprimer que tous les attributs des objets existant à l'état d'après ont un domaine, à défaut d'avoir une valeur si celle-ci n'est pas précisée dans la postcondition du comportement considéré.

Ce choix sémantique sur les ensembles d'instances peut paraître restrictif, car il cherche à minimiser les inconnues dans l'état après. Néanmoins, il comporte quelques bienfaits. En effet, si deux prédicats `\fresh` portent sur des objets potentiellement de même type dynamique, la solution trouvée forcera ces deux objets à être égaux. Pour exprimer que ces objets sont différents, il faudra clairement exprimer ceci dans la postcondition de l'opération. Par exemple, si une postcondition contient le prédicat suivant :

```
\fresh(a1) && \fresh(a2)
```

avec `a1` et `a2` tous deux de type statique `A`, alors il faudra expliciter `a1 != a2` dans la postcondition pour forcer ces deux objets à être différents.

Cette restriction peut permettre de détecter des faiblesses de modélisation avec l'emploi des `\fresh`, comme décrit dans le petit exemple précédent. Nous pensons que demander à l'utilisateur d'expliciter clairement les cas d'*aliasing* peut être un bon moyen

de construire un modèle clair, efficace et exploitable par la suite.

Intéressons-nous à présent aux impacts liés à l'exécution des constructeurs d'objets.

6.4.2 Constructeurs de classe

L'exécution des constructeurs d'objets en deux temps nous impose de recréer ce mécanisme. Nous avons exprimé dans la sous-partie 5.4 le constructeur par défaut d'une classe d'objet. Comme nous l'avons vu, il est nécessaire d'instancier la valeur de l'adresse mémoire associée à l'objet nouvellement créé. De plus, il nous faut recréer le mécanisme de l'appel au constructeur d'objet par défaut avant l'exécution d'un constructeur défini dans la classe.

En se basant sur le mécanisme décrit précédemment, nous décrivons le principe de la primitive $jml\_createInstance(C, M, P, E)$, qui exécute le constructeur M de la classe C, avec les paramètres P dans l'état E et retourne la valeur de l'adresse de la nouvelle instance ainsi obtenue.

1. Exécution systématique de l'opération `constructor`, sans paramètres, en réalisant un appel à la primitive $jml\_executeMethod(C, \texttt{constructor}, [], E)$. Celle-ci retourne la valeur de l'adresse mémoire associée au nouvel objet, et le nouvel environnement d'exécution $E1$.
2. Si le constructeur M appelé est différent du constructeur par défaut, c'est-à-dire s'il est défini par l'utilisateur, la primitive $jml\_executeMethod(C, M, P, E1)$ est ensuite exécutée pour permettre d'établir la postcondition de M.

Cet algorithme active d'abord systématiquement le constructeur par défaut de la classe, avant d'activer le constructeur défini dans la classe, si le constructeur invoqué n'est pas le constructeur par défaut. Nous supposons que le constructeur par défaut ne peut être activé que si aucun autre constructeur de classe n'existe.

Illustrons le fonctionnement décrit dans cette partie sur l'animation de l'exemple.

6.4.3 Animation symbolique de l'exemple

Nous proposons de réaliser une petite séquence d'animation, composée d'une création d'objet et de deux invocations de méthodes, qui permettra d'illustrer les propos décrits dans cette partie. Par souci de concision, les paramètres sont retirés des environnements.

Nous partons de l'état initial donné par les contraintes suivantes :

$$\langle \emptyset \quad , \quad \{H = \{a_1, a_2, ..., a_N\}, I = I_{Purse} = I_{LimitedPurse} = I_{History} = A = T = \emptyset \\ Purse.balance = \emptyset, Purse.hist = \emptyset, History.bal = \emptyset, History.prev = \emptyset, \\ LimitedPurse.max = 10000\}\rangle$$

Nous créons ensuite un nouveau porte-monnaie à l'aide du constructeur défini `Purse(short)`. En java, cette action est définie par :

```
Purse p = new Purse(0);
```

Nous obtenons l'environnement avant/après suivant :

$$\begin{aligned}\langle \emptyset \quad , \quad & \{H = \{a_1, a_2, ..., a_N\}, I = I_{Purse} = I_{LimitedPurse} = I_{History} = A = T = \emptyset,\\ & Purse.balance = \emptyset, Purse.hist = \emptyset, History.bal = \emptyset, History.prev = \emptyset,\\ & LimitedPurse.max = 10000, I' = \{a_1\}, I'_{Purse} = \{a_1\}, I'_{LimitedPurse} = \emptyset, I'_{History} = \emptyset,\\ & A' = \{a_1\}, T' = \{a_1 \mapsto Purse\}, Purse.balance' = \{a_1 \mapsto 0\}, Purse.hist' = \{a_1 \mapsto null\},\\ & History.bal' = \emptyset, History.prev' = \emptyset, LimitedPurse.max' = 10000\}\rangle\end{aligned}$$

On notera la valeur de l'instance nouvellement créée : a_1. Les modifications réalisées au niveau du module d'exécution ont permis d'affecter à la nouvelle instance une valeur physique, au lieu d'une valeur qui aurait été symbolique si les modifications n'avaient pas été effectuées.

Nous invoquons ensuite sur cette nouvelle instance la méthode de crédit avec une valeur de 100. En Java, cette action s'exprime par :

```
p.credit(100);
```

Nous obtenons l'environnement avant/après suivant :

$$\begin{aligned}\langle \emptyset \quad , \quad & \{H = \{a_1, a_2, ..., a_N\}, I = \{a_1\}, I_{Purse} = \{a_1\}, I_{LimitedPurse} = \emptyset, I_{History} = \emptyset, A = \{a_1\},\\ & T = \{a_1 \mapsto Purse\}, Purse.balance = \{a_1 \mapsto 0\}, Purse.hist = \{a_1 \mapsto null\},\\ & History.bal = \emptyset, History.prev = \emptyset, LimitedPurse.max = 10000, I' = \{a_1, a_2\},\\ & I'_{Purse} = \{a_1\}, I'_{LimitedPurse} = \emptyset, I'_{History} = \{a_2\}, A' = \{a_1\}, T' = \{a_1 \mapsto Purse,\\ & a_2 \mapsto History\}, Purse.balance' = \{a_1 \mapsto 100\}, Purse.hist' = \{a_1 \mapsto a_2\},\\ & History.bal' = \{a_2 \mapsto 0\}, History.prev' = \{a_2 \mapsto null\}, LimitedPurse.max' = 10000\}\rangle\end{aligned}$$

Les modifications effectuées permettent d'obtenir un ensemble de nouvelles instances limité à $\{a_2\}$, induit par la création d'un seul nouvel objet `History`, ce qui permet d'obtenir un état suivant clairement défini. On remarque que le nouvel objet créé n'est pas disponible car il n'est pas retourné par la méthode.

Pour terminer, nous invoquons toujours sur le même porte-monnaie une opération de débit, mais sans en préciser la valeur. En Java, cette action s'exprime par :

```
p.debit(_X);
```

où $\_X$ désigne le paramètre symbolique. Nous obtenons alors deux comportements activables. Le premier consiste à effectuer le débit :

$$\begin{aligned}\langle & \{\_X \in 0..100, \_Y \in -32768..32767, \_Y = 100 - \_X\},\\ & \{H = \{a_1, a_2, ..., a_N\}, I = \{a_1, a_2\}, I_{Purse} = \{a_1\}, I_{LimitedPurse} = \emptyset, I_{History} = \{a_2\},\\ & T = \{a_1 \mapsto Purse, a_2 \mapsto History\}, Purse.balance = \{a_1 \mapsto 100\}, Purse.hist = \{a_1 \mapsto a_2\},\\ & History.bal = \{a_2 \mapsto 0\}, History.prev = \{a_2 \mapsto null\}, LimitedPurse.max = 10000,\\ & I' = \{a_1, a_2, a_3\}, I'_{Purse} = \{a_1\}, I'_{LimitedPurse} = \emptyset, I'_{History} = \{a_2, a_3\}, A' = \{a_1\},\\ & T' = \{a_1 \mapsto Purse, a_2 \mapsto History, a_3 \mapsto History\}, Purse.balance' = \{a_1 \mapsto \_Y\},\\ & Purse.hist' = \{a_1 \mapsto a_3\}, History.bal' = \{a_2 \mapsto 0, a_3 \mapsto 100\},\\ & History.prev' = \{a_2 \mapsto null, a_3 \mapsto a_2\}, LimitedPurse.max' = 10000\rangle\end{aligned}$$

Ceci contraint la valeur du paramètre $\_X$ à être dans les limites acceptées (entre 0 et 100 inclus).

Le second comportement consiste à déclencher l'exception et ne pas effectuer le débit. Dans ce cas, aucune valeur n'est modifiée, et la méthode déclenche une exception (non visible ici car elle est modélisée par un paramètre) :

$$\langle \{\_X \in -32768..-1 \cup 101..32767\},$$
$$\{H = \{a_1, a_2, ..., a_N\}, I = \{a_1, a_2\}, I_{Purse} = \{a_1\}, I_{LimitedPurse} = \emptyset, I_{History} = \{a_2\},$$
$$T = \{a_1 \mapsto Purse, a_2 \mapsto History\}, Purse.balance = \{a_1 \mapsto 100\}, Purse.hist = \{a_1 \mapsto a_2\},$$
$$History.bal = \{a_2 \mapsto 0\}, History.prev = \{a_2 \mapsto null\}, LimitedPurse.max = 10000,$$
$$I' = \{a_1, a_2\}, I'_{Purse} = \{a_1\}, I'_{LimitedPurse} = \emptyset, I'_{History} = \{a_2\}, A' = \{a_1\},$$
$$T' = \{a_1 \mapsto Purse, a_2 \mapsto History\}, Purse.balance' = \{a_1 \mapsto 100\}, Purse.hist' = \{a_1 \mapsto a_2\},$$
$$History.bal' = \{a_2 \mapsto 0\}, History.prev' = \{a_2 \mapsto null\}, LimitedPurse.max' = 10000\} \rangle$$

Nous avons illustré par cet exemple le principe décrit dans la partie précédente, lié à la création dynamique d'objets lors de l'exécution des méthodes `credit(short)` et `debit(short)`. Nous présentons à présent le bilan de ce chapitre.

6.5 Synthèse

Nous avons vu dans ce chapitre l'expression des spécifications JML dans le format BZP. Nous avons dans un premier temps donné l'expression des spécifications de types JML, puis nous avons décrit l'expression des spécifications de méthodes. Pour ce faire, nous avons introduit des extensions au format BZP de manière à distinguer les prédicats de spécification JML des prédicats BZP usuels utilisés pour exprimer les structures des classes.

Ce travail a été implanté dans un compilateur réalisant la traduction des fichiers Java/JML vers un fichier au format BZP. Il a donné lieu à une publication dans [BDLU05b] qui a présenté le mécanisme d'animation symbolique des spécifications JML. Un prototype implantant ces principes a également été développé. Il sera présenté en partie III.

La traduction proposée dans ce chapitre donne accès à l'utilisation des outils d'animation et de génération de tests de BZ-Testing-Tools. Néanmoins le passage aux notations objet impose différentes adaptations de l'existant.

Nous présentons dans le chapitre suivant l'application des résultats présentés dans ce chapitre et le précédent à la génération automatique de tests aux limites.

Chapitre 7

Génération de tests aux limites à partir de modèles JML

Sommaire

Ce chapitre décrit la génération automatique de tests pour les programmes Java à partir de modèles formels écrits en JML.

La principale motivation de ce travail est l'utilisation du modèle JML à part entière pour guider la définition de l'objectif de test et la génération des données de test. De plus, les séquences de test produites sont calculées par animation symbolique du modèle, en utilisant l'interprétation des spécifications JML par des systèmes de contraintes, comme décrit dans le chapitre 6.

L'avantage majeur de JML pour la génération de test est sa proximité avec Java. Comme nous l'avons vu précédemment, JML et Java partagent le même niveau d'abstraction. Par conséquent, ils partagent les mêmes attributs et les mêmes méthodes. Cette proximité facilite l'utilisation du modèle pour la génération de test. De plus, le Runtime Assertion Checker de JML fournit un oracle sur les séquences de test. Là où les tests à partir de modèles B ou Statecharts étaient composés d'une partie d'observation, les cas de test JML ne sont composés "que" d'un préambule et d'un corps de test.

Les tests produits visent à mettre en évidence des non-conformités entre la spécification et l'implantation. Nous utilisons ainsi le Runtime Assertion Checker de JML pour détecter les cas de non-conformité lors de l'exécution des cas de test sur l'implantation. Le schéma de validation d'une implantation Java à partir d'une spécification JML est donné en figure 7.1. A partir d'un fichier source Java/JML, nous commençons par analyser et partitionner la spécification, suivant des critères de couverture choisis. Cette étape aboutit à la production de cibles de test qui vont servir à la génération de cas de test Java, suivant des paramètres de génération. Parallèlement, le code source Java enrichi par JML est compilé avec le compilateur `jmlc` qui rajoute la vérification des assertions à la volée. Les cas de test produits sont ensuite passés sur ce byte-code, confrontant l'implantation à la spécification, et produisant un verdict de test : OK si aucune assertion JML n'est violée, ou KO dans le cas contraire.

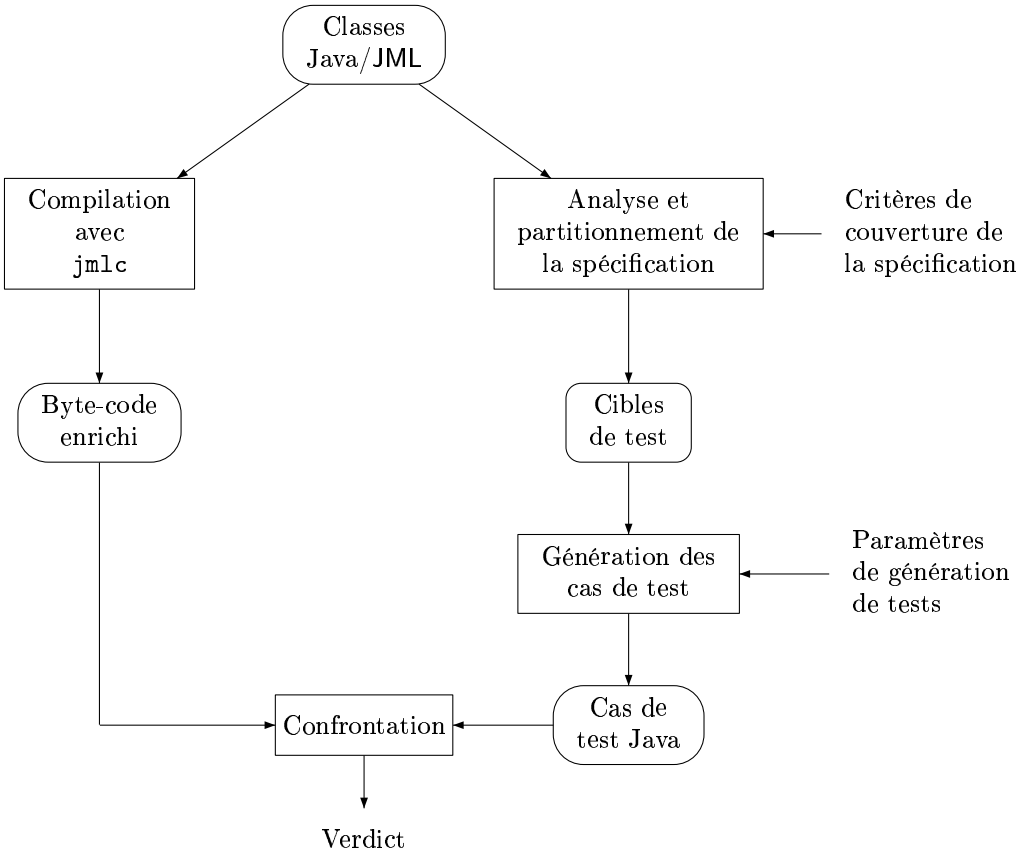


FIG. 7.1 – Schéma de validation d'une programme Java à partir de sa spécification JML

Nous avons mis en place différents moyens pour générer des tests à partir de spécifications JML, que nous allons aborder tout au long de ce chapitre. Le point commun à toutes ces méthodes est l'utilisation de la représentation des spécifications Java/JML par systèmes de contraintes, pour calculer les cibles de test et les séquences de test.

La génération de tests aux limites pour JML est une adaptation des travaux réalisés au LIFC sur la génération de tests aux limites à partir de spécifications B [LPU02] ou

Statecharts [CLL04, Leb05]. Comme nous allons le voir, la génération de tests aux limites à partir de JML emprunte à la couverture de comportements des Statecharts et les valeurs limites pour les variables à domaines ordonnés (en Java/JML, les entiers).

Dans ce chapitre, nous allons, dans un premier temps, définir nos objectifs de test en terme de couverture des comportements, des disjonctions et des données pour les deux types de données supportés, les entiers et les objets. Nous verrons ensuite l'utilisation de la représentation par système de contraintes pour calculer les valeurs des données aux limites. Pour finir, nous verrons le calcul d'une séquence de test à l'aide d'algorithmes de recherches. Tout au long du chapitre, nous illustrons les concepts mis en jeu à l'aide de l'exemple JML présenté au chapitre 4.

7.1 Définition des objectifs de test

L'objectif du test à partir de spécifications JML est de tester les méthodes contenues dans une classe ou un ensemble de classes.

Le test à partir de JML se base sur la génération de tests à partir des comportements extraits des spécifications de méthodes JML.

Cette partie présente la définition des objectifs de test au niveau de la couverture des comportements de la spécification, la couverture des décisions, et la couverture des données que nous avons choisi de mettre en œuvre, dans le contexte de langages de spécification objet.

7.1.1 Couverture des comportements

Tout comme pour le test à partir de spécifications B et Statecharts, l'activation des comportements des spécifications de méthodes définit notre objectif de test.

La nécessité d'une *spécification déterministe* est alors primordiale, puisqu'elle permet de garantir pour une précondition donnée, qu'un seul type de comportement pourra être activé : soit le comportement normal, soit un des comportements exceptionnels. Nous prenons comme hypothèse de travail que les spécifications de méthode que nous traitons sont écrites sous cette forme. Nous présenterons au chapitre suivant une technique permettant de valider, entre autres choses, les spécifications des méthodes JML.

Le graphe des comportements extraits d'une spécification de méthode JML déterministe est donné en figure 7.2. Sur cette figure, chacun des comportements de la spécification de méthode JML correspond à un chemin dans le graphe entre le nœud 1 et le nœud 0.

Différents critères s'offrent à nous pour assurer la couverture de ce graphe.

- Le critère *"tous les nœuds"* sera rempli si le jeu de tests permet l'activation de tous les nœuds du graphe.
- Le critère *"tous les arcs"* sera satisfait si le jeu de tests permet l'activation de tous les arcs indépendemment les uns des autres.
- Le critère *"tous les chemins"* sera rempli si le jeu de test produit permet l'activation de tous les chemins, i.e., les comportements issus du graphe.

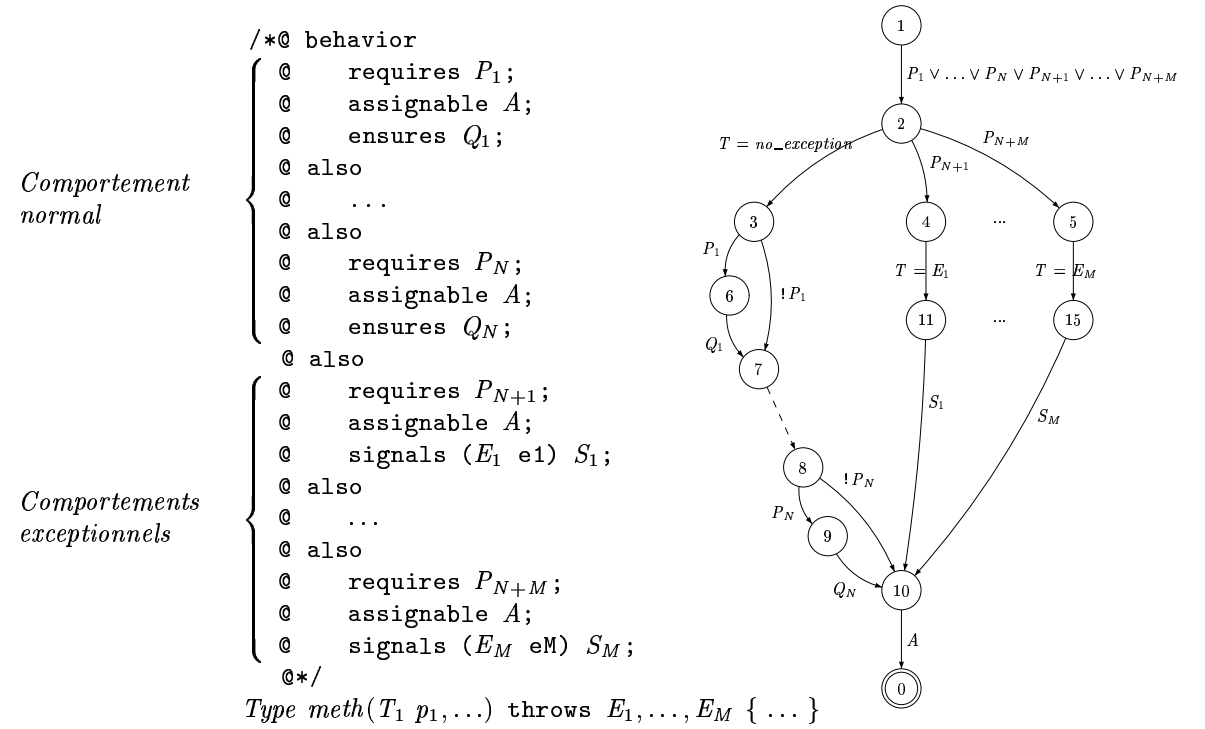


FIG. 7.2 – Extraction des comportements d'une spécification déterministe

La couverture de la spécification est ainsi garantie par la couverture des comportements. Notre approche cherchera à activer chacun des comportements exceptionnels, et à couvrir tous les chemins représentant un comportement normal. Ainsi, nous appliquons le critère de couverture *tous les chemins*.

Dans le cas où des prédicats décrivant le comportement normal sont mutuellement exclusifs, chaque bloc de spécification représente un comportement. Ceci conduit à la génération de comportements inconsistants qui seront donc filtrés lors de la construction du graphe BGP.

Exemple 7.1 (Extraction des comportements) Reprenons l'exemple décrit en partie 4.4.2. Les comportements issus des méthodes de la classe `Purse` sont décrits dans le tableau 7.1.

7.1.2 Atomicité des conditions

La couverture des disjonctions consiste à chercher différents critères pour couvrir les possibilités de satisfaire un prédicat disjonctif. Pour ce faire, notre approche considère les réécritures déjà proposées dans la méthode BZ-TESTING-TOOLS, et décrites en partie 3.5.2. Ces réécritures, proposées pour couvrir les disjonctions présentes dans les décisions, sont réutilisées pour couvrir les différentes parties des prédicats des comportements.

Exemple 7.2 (Réécriture des prédicats disjonctifs) Considérons la précondition du comportement exceptionnel de la méthode `debit(short)` de la classe `Purse` donnée

Méthode	Précondition	Terminaison	Postcondition
`Purse(short)`	`a >= 0`	Normale	`balance == a && hist == null`
`credit(short)`	`a > 0 && \typeof(this) == \type(Purse)`	Normale	`balance == (short)\old(balance) + a && \fresh(hist) && hist.getBalance() == \old(balance) && hist.getPrevious() == \old(hist)`
`debit(short)`	`a > 0 && a <= balance`	Normale	`balance == (short)\old(balance) - a && \fresh(hist) && hist.getBalance() == \old(balance) && hist.getPrevious() == \old(hist)`
	`a <= 0 || a > balance`	`WrongParameter--Exception`	`balance == \old(balance) && hist == \old(hist)`
`getBalance()`	*true*	Normale	`\result == balance`
`transfer(Purse)`	`p != null`	Normale	`balance == p.balance && \fresh(hist) && hist.getBalance() == \old(balance) && hist.getPrevious() == \old(hist)`
`cancel()`	`hist != null`	Normale	`balance == \old(hist.getBalance()) && hist == \old(hist.getPrevious())`

TAB. 7.1 – Comportements extraits de la classe `Purse` de l'exemple

en table 7.1. Les réécritures appliquées à la précondition

```
a <= 0 || a > balance
```

donnent les résultats suivants :

```
                                                                a <= 0 || a > balance     (réécriture 1)
                                                                 a <= 0 [] a > balance    (réécriture 2)
                                                a <= 0 && a <= balance [] a > 0 && a > balance    (réécriture 3)
a <= 0 && a <= balance [] a > 0 && a > balance [] a <= 0 && a > balance    (réécriture 4)
```

On notera que la réécriture 4 produit un cas inconsistant (`a <= 0 && a > balance`) qui ne sera pas considéré par la suite, car il sera filtré.

7.1.3 Couvertures des données

La couverture des données que nous proposons consiste à considérer des valeurs aux limites pour les paramètres et les attributs considérés dans les prédicats d'un comportement réécrit.

Nous obtenons alors deux cas de figure relatifs aux types de données que nous manipulons. Nous définissons d'abord les valeurs limites pour des données numériques.

DÉFINITION 9 (VALEUR NUMÉRIQUE "AUX LIMITES") *Une variable numérique présente une valeur "aux limites" si elle prend une valeur à un extremum (minimum ou maximum) de son domaine dans un contexte donné.*

Pour calculer les valeurs limites des variables numériques, nous considérons une fonction d'optimisation f pour les variables numériques V_{num} [Peu02]. En pratique, cette fonction est la somme des variables numériques, $f = \sum_{v \in V_{num}} v$. Le calcul des valeurs aux limites pour ces attributs numériques *Num* est effectué en minimisant (fonction *minimize*)

ou en maximisant (fonction *maximize*) les valeurs numériques obtenues dans le contexte de l'invariant JML (Inv) et de la partie avant du comportement testé (Cpt_{avant}). La fonction BV^{min} (resp. BV^{max}) ci-dessous donne le calcul des valeurs minimales (resp. maximales) pour un comportement Cpt donné duquel sont extraits les variables numériques V_{num}.

$$BV^{min}(Cpt) = \text{minimize}(f(V_{num}), Inv \wedge Cpt_{avant}) \quad (7.1)$$

$$BV^{max}(Cpt) = \text{maximize}(f(V_{num}), Inv \wedge Cpt_{avant}) \quad (7.2)$$

Nous définissons à présent la notion de valeur limite pour un objet. Cette notion s'appuie sur des valeurs représentant des erreurs classiques de valeurs d'objets. En plus du cas d'utilisation "nominal" (cas 3), notre approche s'intéresse à la possibilité qu'une référence soit `null` (cas 1), réflexive (cas 2), un objet de type hérité (cas 4) ou un alias (cas 5).

DÉFINITION 10 (VALEUR OBJET "AUX LIMITES") *Un objet de type statique T présente une valeur "aux limites" s'il est assigné à une des valeurs suivantes :*

1. `null` *;*
2. `this` *si* `\typeof(this) <: \type(`T`)` *;*
3. *un objet o tel que o* `!= null && `o `!= this && \typeof(`o`) == \type(`T`)` *;*
4. *un objet o tel que o* `!= null && `o `!= this && \typeof(`o`) <: \type(`T`)` *;*
5. *un objet o_i tel que o_i* `==` *o_j et o_j est un autre objet disponible.*

Pour les cas 2, 3, 4 et 5, nous n'excluons pas la possibilité qu'un des attributs de l'objet soit lui-même aux limites.

Les valeurs limites des objets pour chacun des cas peuvent être éliminées si elles provoquent une inconsistance par rapport au comportement considéré. Notre approche consiste à positionner à des valeurs limites les paramètres et les attributs qui interviennent dans le comportement que l'on considère.

Après avoir défini nos objectifs de test à travers les critères de couverture pour les comportements, les disjonctions, et les données contenues dans les spécifications de méthode, nous nous intéressons dans les deux parties suivantes au calcul des cas de test. Nous considérons d'abord par génération des cibles de tests définissant un état symbolique cible pour l'activation du comportement sous test. Puis nous verrons le calcul automatique d'une séquence d'exécution menant à cet état depuis l'état initial.

7.2 Calcul des cibles de test

Ce calcul vise à produire une cible de test sous la forme d'un prédicat de contexte et d'un prédicat de spécialisation.

DÉFINITION 11 *(Cible de test) Une cible de test P_{target} est la conjonction de deux prédicats représentant, pour l'un, le contexte d'activation d'un comportement P_{ctxt}, et pour l'autre, un prédicat de spécialisation P_{spe}.*

$$P_{target} = P_{ctxt} \wedge P_{spe} \quad (7.3)$$

On notera que le prédicat de spécialisation peut-être fourni par un utilisateur, ou calculé automatiquement pour fixer les valeurs des paramètres et des attributs des objets à être aux limites.

7.2.1 Prédicat de contexte

Le prédicat de contexte correspond à l'état dans lequel on souhaite placer le système pour activer le comportement à tester. Ce contexte correspond à la partie avant du comportement, c'est-à-dire à tous les littéraux contenus dans un comportement réécrit qui ne référencent pas des attributs à l'état après.

Le calcul du comportement sous test se base sur l'extraction des comportements à partir des spécifications de méthodes à tester. Chacun de ces comportements subit une réécriture des disjonctions qui produit un nouveau graphe de comportements.

7.2.2 Prédicat de spécialisation

Le calcul du prédicat de spécialisation associé à un comportement testé, ayant été réécrit est décrit par le diagramme en figure 7.3.

Ce calcul se base sur la représentation symbolique des états objets définie dans le chapitre 5. Il s'effectue à partir d'un état symbolique "général" qui regroupe tous les états concrets du système pour un nombre d'instances fini donné. Cet état est obtenu en posant les contraintes correspondant à l'invariant BZP et aux préconditions BZP donnant le typage des paramètres. L'étape suivante consiste à poser les contraintes de la partie avant du comportement testé. Puis les contraintes correspondant aux valeurs limites des objets sont posées, conformément aux valeurs choisies parmi les règles 1 à 5 décrites précédemment. On obtient ainsi un prédicat de spécialisation objet $P_{spe(obj)}$. Les variables numériques, liées aux paramètres, et qui, mises en jeu dans les prédicats du comportement, sont ensuite calculées en minimisant ou en maximisant la valeur obtenue par l'application de la fonction d'optimisation. Ceci produit un prédicat de spécialisation numérique $P_{spe(num)}$ qui se présente sous la forme d'un ensemble d'égalités de type $var = val$ entre une variable numérique (attribut ou paramètre) var et sa valeur val. Pour finir, nous nous assurons de la consistance de l'état symbolique obtenu, en effectuant une instanciation de l'état cible. Si cet état est consistant, alors le prédicat de spécialisation est considéré comme cible à atteindre.

Exemple 7.3 (Calcul des prédicats de spécialisation sur l'exemple) Considérons le comportement normal issu de la méthode `transfer(Purse)` de l'exemple. Nous considérons l'invocation de cette méthode sur un objet de la classe `LimitedPurse`, pour laquelle cette méthode est héritée. L'objectif de test associé à ce comportement est :

$$\texttt{balance >= 0 \&\& balance <= max \&\& p != null \&\}\ P_{spe}$$

Cet objectif est composé de l'invariant de classe de l'objet hôte, de la partie avant du comportement, et du prédicat de spécialisation P_{spe}, donné dans le tableau 7.2. Dans ce tableau, les P_{spe} produisant une cible de test inconsistante sont retirées. R_{num} (resp. R_{obj}) désigne la règle de calcul qui s'applique sur les variables numériques (resp. les variables

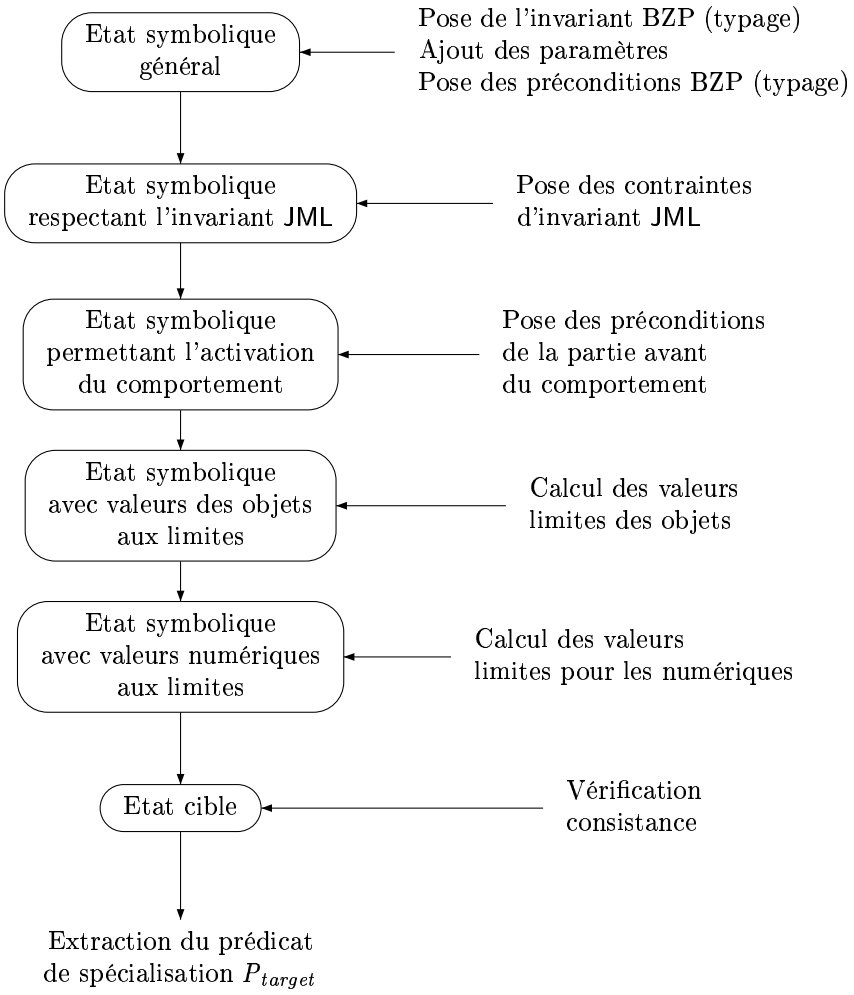


FIG. 7.3 – Schéma du calcul du prédicat P_{spe}

objet). Dans cette méthode, les paramètres et les attributs de l'état avant qui sont mis en jeu sont `this.balance`, `p.balance`, `hist`. Les variables numériques associées aux deux premiers attributs sont minimisés/maximisés pour produire le prédicat de spécialisation, le troisième attribut, `hist`, est également considéré à une valeur limite.

Pour expliciter le calcul des valeurs de test, nous détaillons ici les étapes et chacun des états symboliques permettant d'obtenir le cas (h) du tableau 7.2. Le calcul débute par un état symbolique général ci-dessous, contenant les attributs des classes `Purse` et `LimitedPurse`, les paramètres d'entrée de la méthode, ainsi que les contraintes associées au typage des attributs et des paramètres.

$$H = \{a_1, a_2, \ldots\},\ I \subseteq H,\ I_{Purse} \subseteq I,\ I_{LimitedPurse} \subseteq I,\ I_{LimitedPurse} \subseteq I_{Purse},$$
$$I_{History} \subseteq I,\ I_{History} \cap I_{Purse} = \emptyset,\ T \in I \rightarrow \{Purse, LimitedPurse\},$$
$$Purse.balance \in I_{Purse} \rightarrow -32768..32767,\ Purse.hist \in I_{Purse} \rightarrow I_{History} \cup \{null\},$$
$$LimitedPurse.max \in -32768..32767,\ History.bal \in I_{History} \rightarrow -32768..32767,$$
$$History.prev \in I_{History} \rightarrow I_{History} \cup \{null\},\ this \in I_{LimitedPurse},\ p \in I_{Purse} \cup \{null\}$$

	Prédicat de spécialisation P_{spe}	R_{num}	R_{obj}
(a)	`this.balance == 0 && p == this && hist == null`	minimize	`p` : (2) `hist` : (1)
(b)	`this.balance == 0 && p == this && hist != null && \typeof(hist) = \type(History)`	minimize	`p` : (2) `hist` : (3)
(c)	`this.balance == 10000 && p == this && hist == null`	maximize	`p` : (2) `hist` : (1)
(d)	`this.balance == 10000 && p == this && hist != null && \typeof(hist) = \type(History)`	maximize	`p` : (2) `hist` : (3)
(e)	`this.balance == 0 && p != null && p != this && \typeof(p) == \type(Purse) && p.balance == 0 hist == null`	minimize	`p` : (3) `hist` : (1)
(f)	`this.balance == 0 && p != null && p != this && \typeof(p) == \type(Purse) && p.balance == 0 hist != null && \typeof(hist) == \type(History)`	minimize	`p` : (3) `hist` : (3)
(g)	`this.balance == 10000 && p != null && p != this && \typeof(p) == \type(Purse) && p.balance == 32767 hist == null`	maximize	`p` : (3) `hist` : (2)
(h)	`this.balance == 10000 && p != null && p != this && \typeof(p) == \type(Purse) && p.balance == 32767 hist != null && \typeof(hist) == \type(History)`	maximize	`p` : (3) `hist` : (3)
(i)	`this.balance == 0 && p != null && p != this && \typeof(p) == \type(LimitedPurse) && p.balance == 0 hist == null`	minimize	`p` : (4) `hist` : (1)
(j)	`this.balance == 0 && p != null && p != this && \typeof(p) == \type(LimitedPurse) && p.balance == 0 hist != null && \typeof(hist) == \type(History)`	minimize	`p` : (4) `hist` : (3)
(k)	`this.balance == 10000 && p != null && p != this && \typeof(p) == \type(LimitedPurse) && p.balance == 10000 hist == null`	maximize	`p` : (4) `hist` : (1)
(l)	`this.balance == 10000 && p != null && p != this && \typeof(p) == \type(LimitedPurse) && p.balance == 10000 hist != null && \typeof(hist) == \type(History)`	maximize	`p` : (4) `hist` : (3)

TAB. 7.2 – Prédicats de spécialisation issus de la méthode `transfer(Purse)`

Ce système de contraintes est exprimé à l'aide des conventions définies dans le chapitre 5. Il représente les contraintes initiales portant sur la structure des classes et le typage des attributs. On remarque que les ensembles d'instances de type `Purse` et `History` sont disjoints car héritant tous deux de la classe `Object` (non représentée ici par souci de concision).

Nous posons ensuite les contraintes relatives à l'invariant JML. Par souci de concision, nous ne présentons que les nouvelles contraintes introduites :

$$\forall\, o.o \in I_{Purse} \Rightarrow Purse.balance(o) \geq 0,\ \forall\, o.o \in History.bal(o) \geq 0,$$
$$LimitedPurse.max = 10000,\ \forall\, o.o \in I_{LimitedPurse} \Rightarrow Purse.balance(o) \leq 10000$$

La contrainte suivante est ajoutée pour satisfaire la partie avant du comportement testé.

$$p \neq null$$

ce qui réduit le domaine de p à I_{Purse}. Nous posons désormais les contraintes issues de la génération des valeurs limites pour le paramètre objet `p` suivant la règle n°3 de la définition 10 :

$$p \neq this,\ p \in I_{Purse},\ p \notin I_{LimitedPurse}$$

et les valeurs limites pour l'attribut `hist` de l'objet courant (`this`) d'après la règle n°3 de la définition 10 :

$$Purse.hist(this) \neq null,\ Purse.hist(this) \in I_{History}$$

Pour finir, nous maximisons les variables numériques $Purse.balance(this)$ et $Purse.balance(p)$:

$$\begin{aligned} BV^{max}(cpt_{transfer}) &= maximize(Purse.balance(this) + Purse.balance(p)) \\ &= 42767 \text{ avec } Purse.balance(this) = 10000 \\ &\text{et } Purse.balance(p) = 32767 \end{aligned}$$

Le système de contraintes obtenu étant consistant (i.e., il existe au moins une solution à ces contraintes), nous déduisons donc le prédicat de spécialisation suivant, pour le cas (h), représentant notre prédicat de spécialisation :

```
this.balance == 10000 && p != null && p != this &&
\typeof(p) == \type(Purse) && p.balance == 32767 &&
this.hist != null && \typeof(this.hist) == \type(History)
```

Le calcul des cibles de test étant défini, l'étape suivante consiste à trouver une séquence d'exécution partant de l'état initial et menant à la cible. A chaque cible correspond une séquence d'exécution appelée *cas de test*. La partie suivante présente le calcul ces séquences.

7.3 Calcul de séquences de tests

Une fois la cible de test définie, l'animation symbolique de la spécification est mise en œuvre pour calculer une séquence de créations d'objets et d'invocations de méthodes qui permet d'activer le comportement. Pour atteindre la cible précédemment définie nous employons un algorithme de recherche du "meilleur d'abord" que nous décrivons ci-après. Puis, nous décrivons la phase de réification qui vise à rendre les cas de tests exécutables dans le Runtime Assertion Checker de JML.

7.3.1 Algorithme "meilleur d'abord"

L'algorithme de type "meilleur d'abord" appliqué à la recherche d'une cible de test a été défini par S. Colin dans sa thèse [Col05]. Cet algorithme se base sur un parcours en largeur avec heuristique pour tenter d'atteindre un état cible défini par un prédicat définissant la cible de test. La fonction d'heuristique utilisée cherche à évaluer, pour chaque variable d'état, la "distance" entre la variable d'état de l'état courant et cette même variable d'état dans l'état cible.

Si ces deux variables peuvent s'unifier, c'est-à-dire si ces deux variables peuvent avoir la même valeur, ou plus précisément, si elles partagent un domaine de valeurs commun, la distance entre elles est de 0. Si ce n'est pas le cas, la distance est calculée en fonction du type de donnée (entier, atome, ensemble, ou fonction). Ceci est illustré par la figure 7.4. Dans cette figure, *DomVar* représente le domaine de la variable considérée, *DomVarCour* représente le domaine de la variable de l'état courant (sous-ensemble du domaine de la variable), *DomVarObj* représente le domaine de la variable de l'état cible, et *DomUnif* représente l'intersection entre ces deux derniers domaines, donnant ainsi le sous-ensemble du domaine de la variable qui permet l'unification.

La recherche est lancée depuis l'état initial et se poursuit tant que la cible n'est pas atteinte, i.e., il reste des variables qui ne s'unifient pas. Pour garantir la terminaison du

FIG. 7.4 – Illustration de la distance entre deux variables par rapport à leur domaines

calcul, nous considérons un paramètre qui borne la recherche en profondeur.

Lorsque le calcul se termine par l'atteinte d'un état satisfaisant la cible de test, le calcul de préambule fournit une séquence d'activations de comportements à laquelle on ajoute l'invocation de la méthode sous test pour constituer le cas de test. Contrairement aux cas de test calculés par la méthode BZ-TESTING-TOOLS, les cas de test produits ne présentent pas de phase d'observation, car l'oracle est donné par le passage des tests sur une classe annotée grâce au Runtime Assertion Checker. Par choix, nous avons préféré ne pas considérer de postambule pour enchaîner les tests. Ce choix est motivé par les techniques existantes, basées sur le format JUnit, qui ne considère pas nécessairement de postambule. De plus, l'exécution de tests sur un programme Java/JML est peu coûteuse en temps. La présence d'un postambule, historiquement instauré pour ne pas avoir à systématiquement réinitialiser des systèmes sous test peu "souples", n'est donc plus indispensable.

Une fois les tests abstraits calculés, il est nécessaire de les concrétiser pour produire des cas de test exécutables. C'est le propos de la partie suivante.

7.3.2 Réification des cas de tests abstraits

Comme nous l'avons vu, JML et Java référencent les mêmes méthodes, et les mêmes attributs. Ceci facilite la traduction des cas de test d'abstraits à concrets. Les cas de test abstraits produits se présentent sous la forme d'appels d'opérations BZP qui représentent soit des créations d'objets (si l'opération représente un constructeur), soit des invocations de méthodes (dans le cas contraire).

Nous présentons dans la figure 7.5, la DTD d'un fichier XML qui supporte l'expression des cas de tests abstraits, par la suite réifiés. Un cas de test est composé d'un ensemble d'instructions (nœud *Statement*). Un cas de test est caractérisé par la méthode qui est visée (attributs *class* et *method*) pour un type d'objet (attribut *type*) donné. Il vise à activer un comportement (attribut *behavior*) qui correspond aux nœuds du graphe BGP décrivant le comportement, et une exception (attribut *exception*), qui décrit la terminaison activée par le cas de test. Une instruction (*Statement*) admet un certain nombre de paramètres d'entrées (*Input*) et un éventuel paramètre de sortie (*Output*). Une instruction peut avoir deux genres différents (attribut *kind*). Elle représente soit un appel de

constructeur (*ConstructorCall*), soit un appel de méthode (*MethodCall*). Il est à noter que les noms de classes, méthodes, paramètres et leurs valeurs sont exprimés dans le format BZP.

```
<!ELEMENT TestCase (Statement)+ >
<!ATTLIST TestCase
        class CDATA #REQUIRED
        method CDATA #REQUIRED
        type CDATA #REQUIRED
        behavior CDATA #REQUIRED
        exception CDATA #REQUIRED >

<!ELEMENT Statement (Input+, Output) >
<!ATTLIST Statement
        kind (ConstructorCall | MethodCall) #REQUIRED
        class CDATA #REQUIRED
        method CDATA #REQUIRED >

<!ELEMENT Input EMPTY >
<!ATTLIST Input
        name CDATA #REQUIRED
        value CDATA #REQUIRED >

<!ELEMENT Output EMPTY >
<!ATTLIST Output
        value CDATA #REQUIRED >
```

FIG. 7.5 – Schéma XML d'un cas de test abstrait

Les cas de test produits sont ensuite réifiés [BL03] pour produire du code Java exécutable. Nous supposons l'existence d'un traducteur BZP $\rightarrow$ Java, dénoté par l'opérateur $[\![\_]\!]_{Java}$ qui convertit les noms BZP en leur équivalent Java. L'étape de réification s'effectue de la manière suivante :

- L'ensemble *Statements* des *Statement* est réifié en une instruction Java :

$$[\![Statements]\!]_{Java} \rightsquigarrow [\![Statement_1]\!]_{Java} \ldots [\![Statement_N]\!]_{Java}$$

- Un nœud $Statement_i$ du genre *ConstructorCall* s'exprime par un appel au constructeur représenté par l'attribut *method* de la classe *class* sous la forme d'une expression Java :

$$[\![Statement_i(\texttt{ConstructorCall}, class, method, inputs, output)]\!]_{Java} \rightsquigarrow$$

$$[\![class]\!]_{Java}\ \texttt{var}_i\ \texttt{= new}\ [\![method]\!]_{Java}\texttt{(}[\![inputs]\!]_{Java}\texttt{);}$$

où $\texttt{var}_i$ désigne une variable locale au programme Java qui récupère l'objet créé par l'appel au constructeur. Pour la suite de la réification, une association est conservée

pour enregistrer le lien entre l'instance Java $\texttt{var}_i$ et son équivalent BZP *output*, on notera ce lien par : $\texttt{var}_i \leftrightarrow output$.

- Un nœud $Statement_i$ du genre *MethodCall* qui ne contient pas de valeur de retour (valeur `null`), s'exprime par une invocation de méthode.
  Si la méthode est une méthode statique, alors elle ne s'applique sur aucun objet hôte :

$$[\![Statement_i(\texttt{MethodCall}, class, method^{static}, inputs, \texttt{null})]\!]_{Java} \rightsquigarrow$$

$$[\![class]\!]_{Java}\,.\,[\![method^{static}]\!]_{Java}\,(\,[\![inputs]\!]_{Java}\,)\,;$$

  Si la méthode est une méthode d'instance, elle s'applique sur un objet $\texttt{var}_j$ (associé à la valeur donnée au paramètre nommé *this* de la liste des paramètres *inputs*) :

$$[\![Statement_i(\texttt{MethodCall}, class, method^{inst}, inputs, \texttt{null})]\!]_{Java} \rightsquigarrow$$

$$\texttt{var}_i\,.\,[\![class, method^{inst}]\!]_{Java}\,(\,[\![inputs]\!]_{Java}\,)\,;$$

- Un nœud $Statement_i$ du genre *MethodCall* qui contient une valeur de retour (valeur différente de `null`) dont le type Java est T, se traduit en préfixant aux expressions précédentes le stockage de la valeur de retour dans une variable locale au programme :

$$[\![Statement_i(\texttt{MethodCall}, class, method, inputs, output)]\!]_{Java} \rightsquigarrow$$

$$T\,\texttt{var}_i\ \texttt{=}\ \ldots$$

  et $\texttt{var}_i \leftrightarrow output$.

- Les paramètres d'entrée sont traduits un à un et séparés par des virgules :

$$[\![inputs]\!]_{Java} \rightsquigarrow [\![input_1]\!]_{Java}\,,\ldots,[\![input_N]\!]_{Java}$$

- Chaque paramètre d'entrée peut être :
  - soit une valeur associée à une valeur de retour (*output*) précédent, c'est le cas des objets. La valeur Java associée est donc la variable locale créée pour récupérer cette valeur de retour.

$$[\![o]\!]_{Java} \rightsquigarrow \texttt{var}_j \ \text{ si } \texttt{var}_j \leftrightarrow o$$

  - soit une valeur entière, représentant une paramètre numérique de type Java déclaré T. Dans ce cas, la valeur est transtypée en T pour éviter les erreurs dues à des pertes possibles de précision.

$$[\![num]\!]_{Java} \rightsquigarrow \texttt{(}T\texttt{)}\,num$$

– Pour finir, si une instruction $Statement_i$ est supposée activer un comportement exceptionnel –et déclencher une exception de type E– celle-ci est intégrée dans un mécanisme de rattrapage d'exception. Par extension, nous ajoutons suite à l'instruction une levée d'exception spécifique (de type *JMLTTUnraisedException*) signalant que l'exception attendue n'a pas été déclenchée.

```
try {
    [[Statement_i]]_Java
    throw new JMLTTUnraisedException(''E'');
}
catch (E_exc) {
//  Rien à faire dans ce cas
}
```

Cette exception sera par la suite récupérée par le pilote de test pour compléter le verdict, en signalant qu'un comportement attendu n'a pas été activé.

L'application de ces règles aux cas de test abstraits permet de produire des séquences de test concrétisées, sous la forme d'un programme Java au format JUnit [GHJV94].

Chaque cas de test Java est ensuite exécuté sur une classe compilée avec `jmlc`, enrichie des annotations JML. Le verdict du test est donné par le déclenchement d'exceptions relatives aux assertions JML. Si aucune de ces exceptions n'est déclenchée lors du passage du test, le test a réussi, sinon, il a échoué. L'utilisation du Runtime Assertion Checker permet la vérification des valeurs des attributs dans les postconditions et des propriétés de classe, tels que les invariants ou les contraintes historiques.

Exemple 7.4 (Séquences de test de l'exemple) Reprenons les objectifs de tests calculés pour la méthode `transfer(Purse)` de notre exemple fil rouge, et donnés dans le tableau 7.2. Les séquences de tests calculées pour cette méthode et les verdicts de l'exécution de ces cas de test sont donnés dans le tableau 7.3.

Les cas de test (g) et (h) révèlent une violation de l'invariant JML de la classe `LimitedPurse`. Ce bogue s'explique par le fait que la méthode `transfer(Purse)` a été héritée par la classe `LimitedPurse`, mais n'a pas été redéfinie pour vérifier que la valeur de `balance` respecte l'invariant de `LimitedPurse`. Ainsi, le cas de test proposé permet de détecter une erreur de modélisation objet : l'oubli de redéfinition d'une méthode héritée pour prendre en compte les propriétés de la sous-classe.

On remarque la conception du modèle Java/JML, et notamment la définition d'un constructeur de classe `Purse` et `LimitedPurse` permet d'atteindre aisément les cas de figure où l'attribut `balance` est différent de 0, et l'attribut `hist` est `null`.

Si les constructeurs avaient été définis pour positionner la valeur de l'attribut `balance` à 0 à la construction de l'objet, comme ci-dessous :

```
/*@ assignable balance, hist;
  @ ensures balance == 0 && hist == null;
  @*/
public Purse()  ...
```

Les cibles de test spécifiant les prédicats `this.balance == 10000` et `this.hist == null`

Cible	Cas de test Java	Verdict
(a)	`LimitedPurse var1 = new LimitedPurse((short) 0);` `var1.transfer(var1);`	Succès

(b)	`LimitedPurse var1 = new LimitedPurse((short) 1);`
	`var1.debit((short) 1);`
	`var1.transfer(var1);`
		Succès

(c)	`LimitedPurse var1 = new LimitedPurse((short) 10000);` `var1.transfer(var1);`	Succès

(d)	`LimitedPurse var1 = new LimitedPurse((short) 9999);`
	`var1.credit((short) 1);`
	`var1.transfer(var1);`
		Succès

(e)	`LimitedPurse var1 = new LimitedPurse(0);`
	`Purse var2 = new Purse((short) 0);`
	`var1.transfer(var2);`
		Succès

(f)	`LimitedPurse var1 = new LimitedPurse((short) 1);`
	`var1.debit((short) 1);`
	`Purse var2 = new Purse((short) 0);`
	`var1.transfer(var2);`
		Succès

(g)	`LimitedPurse var1 = new LimitedPurse((short) 10000);`
	`Purse var2 = new Purse((short) 32767);`
	`var1.transfer(var2);`
		Echec → `JMLInvariantError`

(h)	`LimitedPurse var1 = new LimitedPurse((short) 9999);`
	`var1.credit((short) 1);`
	`Purse var2 = new Purse((short) 32767);`
	`var1.transfer(var2);`
		Echec → `JMLInvariantError`

(i)	`LimitedPurse var1 = new LimitedPurse((short) 0);`
	`LimitedPurse var2 = new LimitedPurse((short) 0);`
	`var1.transfer(var2);`
		Succès

(j)	`LimitedPurse var1 = new LimitedPurse((short) 1);`
	`var1.debit((short) 1);`
	`LimitedPurse var2 = new LimitedPurse((short) 0);`
	`var1.transfer(var2);`
		Succès

(k)	`LimitedPurse var1 = new LimitedPurse((short) 10000);`
	`LimitedPurse var2 = new LimitedPurse((short) 10000);`
	`var1.transfer(var2);`
		Succès

(l)	`LimitedPurse var1 = new LimitedPurse((short) 9999);`
	`var1.credit((short) 1);`
	`LimitedPurse var2 = new LimitedPurse((short) 10000);`
	`var1.transfer(var2);`
		Succès

TAB. 7.3 – Cas de tests produits à partir de la méthode `transfer(Purse)`

n'auraient pas pu être atteintes, car elles requièrent au moins une opération sur le porte-monnaie courant pour positionner la valeur du solde, ce qui force l'attribut `hist` à être forcément non `null`. Pour éviter ce cas de figure, l'invariant de la classe `Purse` doit être renforcé pour statuer que `(hist == null) ==> (balance == 0)`, ce qui permet, lors du calcul des cibles de test de filtrer les comportements inconsistants.

Le lecteur intéressé trouvera les cibles de test et les cas de test associés à l'exemple fil rouge en annexe A.

7.4 Synthèse

Nous avons vu dans ce chapitre la génération de tests à partir des modèles JML. Nous avons défini nos objectifs de test, qui sont l'activation des comportements extraits des spécifications des méthodes JML.

Pour calculer les données de test, nous considérons que les paramètres doivent être "aux limites", ce qui nous a amené à définir la notion d'objets aux limites. Ces paramètres aux limites sont intégrés au calcul de préambule des cas de test qui permettent d'activer les méthodes à tester. Pour finir, ces cas de test abstraits sont concrétisés en programme Java permettant l'exécution directe des cas de test et l'obtention immédiate d'un verdict grâce à la vérification d'assertions à la volée de JML.

Un prototype de génération de tests aux limites à partir de JML a été implanté, réalisant la chaîne complète décrite dans ce chapitre. Cet outil, implantant également les contributions relatives à l'animation symbolique de modèles JML est décrit en partie III.

Comme nous l'avons présenté, la génération de tests nécessite un modèle qui soit, d'une part, suffisamment complet dans les postconditions pour être animé et, d'autre part, qu'il n'y ait aucune ambiguïté dans les descriptions des comportements et des propriétés d'état. Nous proposons dans le chapitre suivant une méthode pour la validation des modèles JML destinés à la génération de test, permettant de nous assurer de ces éléments pré-requis.

Chapitre 8

Validation des modèles JML par évaluation symbolique

Sommaire

Au chapitre précédent, nous avons présenté une méthode de génération de tests basée sur un modèle formel, sur lequel nous avons fait certaines hypothèses, comme par exemple l'exclusion mutuelle lors de l'activation des différents types de comportements issus d'une spécification de méthode. La représentation symbolique des états Java, et l'animation symbolique du modèle nous permettent de procéder à la détection de propriétés sur un modèle formel, dans le but d'aider à la conception d'un modèle efficacement exploitable pour la génération de tests. Pour ce faire, le modèle doit satisfaire un certain nombre de propriétés statiques ou dynamiques. Celles-ci sont exprimées par rapport aux états décrits dans les propriétés du système telles que l'invariant, mais également dans les transitions. Nous proposons dans ce chapitre une approche consistant à détecter de manière automatique si le modèle remplit les propriétés désirées.

L'objectif est de mettre en place une méthodologie de vérification de diverses propriétés sur le modèle, permettant ainsi de contrôler sa validité dans le but d'une utilisation

visant la génération de tests. Nous nous attacherons à ce que les vérifications que nous proposons puissent être entièrement automatisées par des mécanismes de résolution de contraintes ou des algorithmes d'animation symbolique. Ainsi, nos vérifications se décomposent en deux catégories : les vérifications de consistance du modèle et la validation pour la génération de tests. Si la première catégorie détecte des erreurs de modélisation qu'il est impératif de corriger, la seconde n'adresse que des avertissements sur la faiblesse du modèle, que l'utilisateur est libre de corriger ou pas. Les travaux présentés dans ce chapitre sont similaires à des approches de consistance de modèles décrivant des besoins [HJL96] ou destinées au test [PK04]. Les détections réalisées sont effectuées par résolution de contraintes ou par une approche de type model-checking borné [BGL+00].

Ce chapitre présente l'application directe de la représentation de JML par systèmes de contraintes décrites aux chapitres 5 et 6 et permet de faire le lien avec les travaux présentés au chapitre 7. Nous commençons par voir les définitions des différents concepts et notations que nous allons employer tout au long de ce chapitre. Nous caractérisons ensuite les propriétés intéressantes à contrôler sur un modèle Java/JML, dans l'objectif d'utiliser ce modèle pour la génération de tests. Nous terminons par l'utilisation de la résolution de contraintes et de l'animation pour procéder aux détections des propriétés précédemment énoncées.

8.1 Définitions

Cette partie définit les différents concepts relatifs à l'animation d'un modèle et à la mise en œuvre de l'animation symbolique. Nous présentons ici des définitions relatives aux modèles exprimés en Java/JML. Nous appliquons ensuite ces définitions et ces concepts aux systèmes décrits par des objets et spécifiés en prenant en compte la notion de conception par contrat.

Nous commençons par considérer un système comme un ensemble de variables dont les valeurs décrivent les différents états du système. Nous définissons ainsi des notions de base qui nous servirons par la suite. Nous introduisons ensuite une restriction sur ces états "généraux" pour pouvoir utiliser les spécificités des notations objets. Ceci nous permet ainsi de considérer des états d'un programme Java/JML et leur représentation sous la forme d'états symboliques.

8.1.1 Etats concrets et transitions d'un système

Nous définissons ici les états et les transitions d'un système, ainsi que les propriétés qui leur sont associées. Ces définitions donnent les bases à l'animation énumérative (par opposition à l'animation symbolique), qui est à la base des model-checkers pour la construction du graphe d'atteignabilité complet d'un système. Nous nous appuierons sur ces définitions pour décrire le passage de l'animation énumérative à l'animation symbolique. Nous commençons par définir ce qu'est un état du système.

DÉFINITION 12 (ETAT DU SYSTÈME) *Soit $\mathcal{V}$ l'ensemble des variables du système, et $\mathcal{D}_\mathcal{V}$ l'ensemble des domaines des variables de $\mathcal{V}$. Un état du système correspond à une valuation possible des variables d'état de ce système. L'ensemble des états du système est noté $\mathcal{S}$, défini par :*

$$\mathcal{S} = \mathcal{V} \rightarrow \mathcal{D}_\mathcal{V} \tag{8.1}$$

Dans toute cette partie, nous désignerons par $P(s)$ la valeur de vérité du prédicat P à l'état s considéré, avec $s \in \mathcal{S}$. Définissons à présent l'état initial d'un système.

DÉFINITION 13 (ETAT INITIAL) *On appelle état initial du système l'état défini par la valuation initiale des variables d'état du système.*

On notera par *Init* les propriétés initiales du système. Par conséquent, $Init(s)$ désigne le fait que l'état s est un état initial.

PROPRIÉTÉ 13 (EXISTENCE D'UN ÉTAT INITIAL) *On suppose l'existence d'au moins un état initial au système. Par conséquent l'ensemble des états initiaux est un sous-ensemble des état du système.*

$$\exists\, s \;.\; s \in \mathcal{S} \wedge Init(s) \tag{8.2}$$

Intéressons-nous désormais aux transitions entre les états du système.

DÉFINITION 14 (TRANSITION) *Les* transitions *du système sont décrites sous la forme de comportements, exprimés par des prédicats avant-après qui doivent être satisfaits entre deux états du système, et mettant en jeu un ensemble de variables locales au comportement, dont on supposera que le domaine est spécifié dans le comportement.*

Une transition est un constructeur d'objet ou une méthode qui possède le plus souvent des paramètres d'entrée, et se compose de préconditions, les conditions d'exécution de l'opération, et de postconditions sous la forme d'un prédicat "avant-après" donnant des propriétés sur l'état suivant en fonction de l'état précédent.

DÉFINITION 15 (PRÉCONDITION D'UNE MÉTHODE/D'UN COMPORTEMENT) *La précondition d'une méthode, notée Req, est la disjonction des prédicats contenus dans les clauses* `requires` *d'une spécification de méthode Java en JML. La précondition d'un comportement, notée Cpt_{avant}, est la partie avant du comportement (extraite de manière syntaxique).*

On notera $Cpt_{avant}(\rho)(s)$ le prédicat correspondant à l'évaluation de la précondition du comportement Cpt, admettant les paramètres ρ à l'état s du système. De manière similaire, on notera $Cpt_{apres}(\rho)(s, s')$ le prédicat correspondant à l'évaluation de la partie après du comportement Cpt mettant en relation les états s et s'. La notation $s_1 \overset{Cpt(\rho)}{\rightarrow} s_2$ désigne l'activation du comportement Cpt à partir de l'état s_1 et menant à l'état s_2.

On note par [] l'opérateur qui désigne un choix entre comportements. Ainsi, chaque opération peut se décomposer sous la forme d'un ensemble de couples $(Cpt_{avant}, Cpt_{apres})$

spécifiant un ensemble de comportements. Par conséquent, une opération $Op(\rho)$ peut être réécrite sous la forme :

$$Op(\rho) \equiv (Cpt^1_{avant}, Cpt^1_{apres})(\rho) \,[]\, \ldots \,[]\, (Cpt^N_{avant}, Cpt^N_{apres})(\rho) \tag{8.3}$$

Illustrons ces définitions avec un exemple simple.

Exemple 8.1 (Précondition d'une méthode/d'un comportement) Considérons la spécification de méthode suivante :

```
/*@ requires x >= 0;
  @ assignable y;
  @ ensures x >= 100 ==> y == x;
  @*/
void meth(short x) { ... }
```

Cette méthode présente deux comportements (extraits en calculant la forme normale disjonctive de l'implication) :

$$\texttt{x >= 0 \&\& x >= 100 \&\& y' == x} \;[]\; \texttt{x >= 0 \&\& x < 100 \&\& y'} \in dom(\texttt{y})$$

La précondition de la méthode est donnée par `x >= 0`. La précondition du premier comportement est `x >= 0 && x >= 100`, et la seconde est `x >= 0 && x < 100`. On notera que dans ce second comportement, la valeur après de `y` n'étant pas spécifiée, `y'` peut prendre n'importe quelle valeur de son domaine de définition (exprimé par $dom(\texttt{y})$).

Les opérations et les comportements sont soumis au principe de la conception par contrat, qui stipule que le système doit remplir la précondition de l'opération pour pouvoir l'exécuter.

PROPRIÉTÉ 14 (SATISFACTION DU CONTRAT D'ACTIVATION) *Un comportement exécuté à l'état s dont les préconditions ne sont pas vérifiées laisse le système dans ce même état.*

$$\forall\, s.s \in \mathcal{S} \Rightarrow (\forall\, \rho.\neg\, Cpt_{avant}(\rho)(s) \Rightarrow s \overset{Cpt}{\rightarrow} s) \tag{8.4}$$

Nous définissons à présent des propriétés relatives aux comportements, en terme de consistance, d'inconsistance, d'activabilité, et d'inactivabilité.

DÉFINITION 16 (COMPORTEMENT CONSISTANT ET INCONSISTANT) *Un comportement $(Cpt_{avant}, Cpt_{apres})$ est dit consistant (resp. inconsistant) lorsque $Cpt_{avant} \wedge Cpt_{apres}$ peut être satisfait entre deux états du système (resp. n'est satisfait entre aucun des états du système). Une transition consistante se formalise par :*

$$\exists\, s, s', \rho \;.\; s, s' \in \mathcal{S} \;\wedge\; Cpt_{avant}(\rho)(s) \wedge Cpt_{apres}(\rho)(s, s') \tag{8.5}$$

Une transition inconsistante se formalise par :

$$\neg\, \exists\, s, s', \rho \;.\; s, s' \in \mathcal{S} \;\wedge\; Cpt_{avant}(\rho)(s) \wedge Cpt_{apres}(\rho)(s, s') \tag{8.6}$$

qui se simplifie par :

$$\forall\, s, s', \rho.s, s' \in \mathcal{S} \Rightarrow \neg(Cpt_{avant}(\rho)(s) \wedge Cpt_{apres}(\rho)(s, s')) \tag{8.7}$$

Définissons à présent la notion de comportement activable et inactivable.

DÉFINITION 17 (COMPORTEMENT ACTIVABLE ET INACTIVABLE) *Un comportement* $(Cpt_{avant}, Cpt_{apres})$ *est dit activable (resp. inactivable) lorsque* Cpt_{avant} *est satisfaisable sur au moins un état (resp. insatisfaisable sur tous les états) du système. Une transition activable se formalise par :*

$$\exists\, s, \rho\ .\ s \in \mathcal{S}\ \wedge\ Cpt_{avant}(\rho)(s) \tag{8.8}$$

Une transition inactivable se formalise par :

$$\neg\, \exists\, s, \rho\ .\ s \in \mathcal{S}\ \wedge\ Cpt_{avant}(\rho)(s) \tag{8.9}$$

qui se simplifie par :

$$\forall\, s, \rho . s \in \mathcal{S} \Rightarrow \neg\, Cpt_{avant}(\rho)(s) \tag{8.10}$$

Les définitions des états du système, consistant à associer à une variable une valeur de son domaine de définition, sont très générales, et ne sont pas forcément en adéquation avec la modélisation d'un système objet décrit en Java/JML. Considérer les domaines des variables est trop large et ne permet pas de décrire, pour la modélisation que nous avons choisie, un système objet. En effet, certaines contraintes d'intégrité du système objet s'appliquent entre les variables du système (comme par exemple, l'inclusion des ensembles d'instances des classes lorsqu'il y a présence d'héritage), et ne sont pas prises en compte dans cette définition.

Nous introduisons donc un *filtrage* des états du système pour ne considérer que les états qui représentent, dans la modélisation choisie, un état objet.

8.1.2 Filtrage des états objets

Cette sous-partie s'intéresse plus particulièrement à restreindre la notion d'état d'un système pour ne considérer que des états représentant réellement des états de programme Java. En effet, ne considérer que les domaines des variables du programme n'exprime que le typage des attributs des différentes classes, indépendamment de la hiérarchie des classes, et les contraintes de disjonctions des ensembles d'instances de différentes sous-classes entre elles.

Si l'on considère deux classes C_1 et C_2, avec la relation d'héritage $C_1 \sqsubseteq_1 C_2$, notre modélisation considère deux ensembles d'instances pour chacune de ces classes I_{C_1} pour C_1 et I_{C_2} pour C_2. Chacune de ces variables est un sous-ensemble du tas H ; le domaine de chacune de ces variables est donc $dom(I_{C_1}) = dom(I_{C_2}) \in \mathbb{P}(H)$. De plus, une contrainte d'intégrité sur la structure hiérarchique des classes s'applique et ainsi $I_{C_1} \subseteq I_{C_2}$.

En se basant sur la représentation symbolique définie au chapitre 5, nous définissons les états d'un système objet comme étant l'ensemble des états respectant des contraintes de structures (liens d'héritages entre les classes) en plus du typage des attributs. Nous désignons par I_{Typ} l'invariant dit "de typage" des données représentées dans un état objet. Cet invariant est exprimé dans la représentation présentée au chapitre 5 par un prédicat

BZP du genre `invariant`.

L'ensemble des états du système est défini par les états qui vérifient l'invariant de typage des données.

DÉFINITION 18 (ETATS OBJETS DU SYSTÈME) *Soit $\mathcal{S}$ l'ensemble des états du système, et soit I_{Typ} l'invariant qui décrit la structure et le typage des données. L'ensemble des états objets du système, noté $\mathcal{S}_o$, est défini par :*

$$\mathcal{S}_o = \{s \mid s \in \mathcal{S} \wedge I_{Typ}(s)\} \tag{8.11}$$

Informellement, $\mathcal{S}_o$ est l'ensemble des états qui satisfont les contraintes de structuration définies dans I_{Typ}.

Nous définissons à présent les états symboliques et l'animation symbolique en tant que transitions entre deux états symboliques.

8.1.3 Etats symboliques et animation symbolique

L'animation symbolique ne considère plus, contrairement à précédemment, une énumération complète des états, mais se focalise sur des états qui sont regroupés dans un ensemble de classes d'équivalences. Ainsi, les états symboliques se définissent comme un sous-ensemble non-vide des états du système. Nous définissons ici le lien entre états symboliques et états concrets, et le lien avec les systèmes de contraintes.

Etats symboliques et animation symbolique

Nous commençons par présenter les états symboliques et l'animation symbolique.

DÉFINITION 19 (ETAT SYMBOLIQUE DU SYSTÈME) *Soit $\mathcal{S}_o$ l'ensemble des états du système objet. Un état symbolique du système, noté s_c, est défini par :*

$$s_c \subseteq \mathcal{S}_o \tag{8.12}$$

L'activation d'un comportement à partir d'un état symbolique réduit les domaines des variables par ajout de contraintes dans le système de contraintes associé à l'environnement. Ainsi, pour un état contraint s_{c_1}, seulement un sous-ensemble de valeurs est consistant avec le comportement, ce sous-ensemble forme un état contraint s_{c_2} inclus dans ce dernier. Si cet état contraint n'est pas vide, alors le comportement est activable et l'ensemble d'activation de ce comportement pour l'état contraint s_{c_1} est s_{c_2}.

DÉFINITION 20 (DOMAINE D'APPLICATION D'UN COMPORTEMENT) *Soit un état symbolique s_{c_1} et un comportement Cpt décomposé en une partie avant et une partie après $(Cpt_{avant}, Cpt_{apres})$. Le domaine d'application de Cpt est l'état contraint s_{c_2} tel que :*

$$s_{c_2} = \{s \mid s \in s_{c_1} \wedge \exists \rho \,.\, Cpt_{avant}(\rho)(s)\} \tag{8.13}$$

Ce domaine d'application d'un comportement est un sous-ensemble de l'état contraint à partir duquel on souhaite activer le comportement.

L'activation d'un comportement, à partir d'un état contraint, mène à un nouvel état contraint.

DÉFINITION 21 (ACTIVATION D'UN COMPORTEMENT) *Soit Cpt un comportement activable depuis un état symbolique s_{c_1}, si le comportement est consistant, le système peut être placé dans n'importe quel état de s_{c_2} défini tel que :*

$$s_{c_2} = \{s' \mid s' \in \mathcal{S}_o \wedge s \in s_{c_1} \wedge \exists\, \rho \;.\; s \overset{Cpt(\rho)}{\rightarrow} s'\} \tag{8.14}$$

Vérification de propriétés

Désormais les propriétés vérifiées ne seront plus considérées sur un état mais sur un ensemble d'états. Ainsi, on distingue non plus deux conclusions sur ces propriétés (vraies ou fausses) mais quatre conclusions possibles : valide, non-valide, satisfaisable, et insatisfaisable. Nous décrivons à présent ces concepts.

DÉFINITION 22 (VALIDITÉ D'UNE PROPRIÉTÉ) *Soit s_c un état contraint du système, et soit Prop une propriété sur les variables du système. Prop est dite valide sur les états contraints du s_c si et seulement si*

$$\forall\, s.s \in s_c \Rightarrow Prop(s) \tag{8.15}$$

DÉFINITION 23 (NON-VALIDITÉ D'UNE PROPRIÉTÉ) *Soit s_c un état contraint du système, et soit Prop une propriété sur les variables du système. Prop est dite non-valide sur les états contraints du s_c si et seulement si*

$$\neg(\forall\, s.s \in s_c \Rightarrow Prop(s)) \tag{8.16}$$

qui se simplifie en :

$$\exists\, s \;.\; s \in s_c \;\wedge\; \neg Prop(s) \tag{8.17}$$

DÉFINITION 24 (SATISFAISABILITÉ D'UNE PROPRIÉTÉ) *Soit s_c un état contraint du système, et soit Prop une propriété sur les variables du système. Prop est dite satisfaisable sur les états contraints du s_c si et seulement si*

$$\exists\, s \in s_c \;.\; Prop(s) \tag{8.18}$$

DÉFINITION 25 (INSATISFAISABILITÉ D'UNE PROPRIÉTÉ) *Soit s_c un état contraint du système, et soit Prop une propriété sur les variables du système. Prop est dite insatisfaisable sur les états contraints du s_c si et seulement si*

$$\neg(\exists\, s \;.\; s \in s_c \;\wedge\; Prop(s)) \tag{8.19}$$

qui se simplifie en :

$$\forall\, s.s \in s_c \Rightarrow \neg Prop(s) \tag{8.20}$$

Ces définitions nous permettent d'exprimer les propriétés que nous souhaitons voir vérifiées sur les modèles Java/JML. Nous décrivons à présent ces propriétés.

8.2 Caractérisation des propriétés à détecter

Nous entendons utiliser la résolution de contraintes et l'animation symbolique pas uniquement pour vérifier le modèle, mais également pour s'assurer qu'il présente les propriétés nécessaires pour être employé dans un processus de génération de tests.

Nous caractérisons dans cette partie les propriétés que nous souhaitons voir vérifiées sur les modèles objet Java/JML. Elles sont données par le tableau 8.1. Ces propriétés se décomposent en deux types : les propriétés de consistance générale d'un modèle objet Java/JML, et les propriétés relatives à la validation de ce modèle dans l'objectif de générer des cas de tests fonctionnels.

Cohérence du modèle	Validation pour le test
– Préservation de l'invariant – Préservation des contraintes historiques – Etablissement des contraintes initiales	– Consistance de l'invariant – Consistance des contraintes historiques – Déterminisme des comportements – Détection des préconditions trop fortes – Détection des préconditions trop faibles – Consistance des comportements

TAB. 8.1 – Propriétés à établir sur le modèle

8.2.1 Vérification de la cohérence du modèle

Nous caractérisons ici les propriétés de consistance que nous voulons établir sur le modèle Java/JML. Nous les définissons en fonction des propriétés JML auxquelles elles se rapportent. Ces dernières ont été définies dans le chapitre 4 et isolées par notre représentation pour permettre d'en disposer aisément.

Contraintes initiales

Nous souhaitons nous assurer qu'il existe au moins un état du système qui satisfait les contraintes initiales, suite à la création d'un objet. Ainsi, l'exécution d'un comportement extrait d'un constructeur permet toujours d'établir les contraintes initiales.

DÉFINITION 26 (VÉRIFICATION DES CONTRAINTES INITIALES) *Soit $CPT^{new}(C)$ l'ensemble des comportements extraits des spécifications des constructeurs d'objets de la classe C. Soient $Initially_C(s, o)$ les contraintes initiales associées à la classe C portant sur un objet o de type C fraîchement créé et s'appliquant dans l'état s. Pour chaque classe C, nous souhaitons établir que pour chaque $Cpt(\rho) \in CPT^{new}(C)$ créant l'objet o :*

$$\exists\, s, s'.s, s' \in \mathcal{S}_o \wedge s \overset{Cpt(\rho)}{\rightarrow} s' \ \wedge \ Initially_C(s', o) \tag{8.21}$$

si l'on veut détecter la satisfaisabilité des contraintes initiales, ou

$$\forall\, s, s'.s, s' \in \mathcal{S}_o \wedge s \overset{Cpt(\rho)}{\rightarrow} s' \Rightarrow Initially_C(s', o) \tag{8.22}$$

si l'on veut détecter la validité des contraintes initiales.

Invariant

Un invariant JML est dit consistant si aucune des propriétés requises sur les instances des classes n'interfèrent entre elles. Il s'agit d'une propriété importante, car il est possible que l'invariant spécifié soit inconsistant, ce qui a pour conséquence de systématiquement et trivialement satisfaire les obligations de preuve d'invariant.

Nous sommes en mesure de définir les propriétés relatives aux invariants, qui sont : la consistance de l'invariant, et sa cohérence vis-à-vis de l'état initial du système et des comportements activables.

DÉFINITION 27 (CONSISTANCE DE L'INVARIANT) *Soit AllC l'ensemble des classes considérées. Soit $Inv_C(s, o)$ l'invariant de la classe C appliqué à l'objet o dans l'état s. Nous souhaitons ainsi établir qu'il existe un état du système qui vérifie l'invariant global. Formellement,*

$$\exists\, s \;.\; s \in \mathcal{S}_o \;\wedge \bigwedge_{C \in AllC} \forall\, o.o \in I_C \Rightarrow Inv_C(s, o) \tag{8.23}$$

L'invariant de classe doit être valide à la création d'un objet, et préservé par l'exécution d'une méthode. Nous souhaitons donc effectuer cette vérification.

En JML, les invariants sont *assumés* lorsqu'ils doivent être vérifiés dans la précondition d'une méthode, *établis* lorsqu'ils doivent être vérifiés par la postcondition d'une méthode, ou *préservés* lorsqu'ils doivent être à la fois assumés et établis. Ainsi, dans notre approche basée uniquement sur le modèle JML, les invariants doivent être établis par la création d'un objet, et doivent être préservés par l'exécution d'une méthode.

DÉFINITION 28 (VÉRIFICATION DE L'INVARIANT) *Soit $CPT(C)$ l'ensemble des comportements extraits des spécifications de méthodes de la classe C. Soient Inv_C les invariants associés à la classe C. Nous souhaitons vérifier que l'exécution d'un comportement d'une méthode préserve l'invariant. Pour chaque classe C, et chaque comportement $Cpt(\rho) \in CPT(C)$, nous souhaitons avoir :*

$$\forall\, s, s'.s, s' \in \mathcal{S}_o \wedge (\bigwedge_{C \in AllC} Inv_C(s)) \wedge s \overset{Cpt(\rho)}{\rightarrow} s' \Rightarrow \bigwedge_{C \in AllC} Inv_C(s')) \tag{8.24}$$

si nous voulons détecter la validité des invariants à l'état d'après, ou :

$$\exists\, s, s' \;.\; s, s' \in \mathcal{S}_o \wedge (\bigwedge_{C \in AllC} Inv_C(s)) \wedge s \overset{Cpt(\rho)}{\rightarrow} s' \wedge \bigwedge_{C \in AllC} Inv_C(s')) \tag{8.25}$$

si nous voulons détecter la satisfaisabilité des invariants à l'état après.

Lorsque l'invariant est violé par l'exécution d'un comportement, il peut être pertinent, dans le but d'aider au débogage du modèle, de rechercher une trace d'exécution menant au contre-exemple. Cette recherche est également envisagée par notre approche et sera décrite comme une détection dynamique.

Contraintes historiques

Les propriétés vérifiées sur les contraintes historiques sont de l'ordre de la consistance de ces contraintes et la validité de ces contraintes par exécution des comportements du modèle.

DÉFINITION 29 (CONSISTANCE DES CONTRAINTES HISTORIQUES) *Soit AllC l'ensemble des classes Java/JML considérées. Soit HC_C les contraintes historiques de la classe C. Nous souhaitons ainsi établir qu'il existe des couples d'états avant-après, qui satisfont la contrainte historique, pour chaque classe C et chaque $Cpt(\rho) \in CPT(C)$.*

$$\exists\, s, s'.s, s' \in \mathcal{S}_o \ \wedge \ s \overset{Cpt(\rho)}{\rightarrow} s' \ \wedge \ HC_C(s, s') \tag{8.26}$$

Nous nous intéressons à présent à la consistance des contraintes historiques et des comportements.

DÉFINITION 30 (VÉRIFICATION DES CONTRAINTES HISTORIQUES) *Soit AllC l'ensemble des classes Java/JML considérées. Soit HC_C les contraintes historiques de la classe C. Nous souhaitons ainsi établir, pour chaque comportement Cpt extraits des spécifications de méthodes, que tous les couples d'états avant-après satisfont la contrainte historique. Formellement, nous cherchons à vérifier, pour chaque classe C, et chaque $Cpt(\rho) \in CPT(C)$:*

$$\exists\, s, s' \ .\ s, s' \in \mathcal{S}_o \ \wedge \ s \overset{Cpt(\rho)}{\rightarrow} s' \wedge HC_C(s, s') \tag{8.27}$$

si nous voulons détecter la satisfaisabilité des contraintes historiques par rapport aux comportements, ou :

$$\forall\, s, s'.s, s' \in \mathcal{S}_o \ \wedge \ s \overset{Cpt(\rho)}{\rightarrow} s' \Rightarrow HC_C(s, s') \tag{8.28}$$

si nous voulons détecter la validité des contraintes historiques par rapport aux comportements.

Les propriétés que nous venons de définir garantissent la cohérence du modèle. Néanmoins, de telles vérifications ne sont pas nécessairement suffisantes en vue d'utiliser le modèle pour la génération de tests. Nous définissons à présent les propriétés que nous souhaitons voir sur le modèle pour le rendre exploitable dans le processus de génération de tests.

8.2.2 Validation du modèle pour la génération de tests

Nous proposons ici des propriétés qui visent à améliorer le modèle Java/JML en vue de l'utiliser dans un processus de génération de tests. Nous verrons ainsi les propriétés relatives à la détection du non-déterminisme des comportements d'une spécification de méthode JML, l'impossibilité d'activer des comportements à cause de préconditions trop fortes, et les imprécisions liées aux préconditions et postconditions jugées trop faibles.

Déterminisme des comportements

Tout d'abord, nous entendons détecter l'exclusion mutuelle des comportements du modèle dans l'objectif d'avoir des comportements déterministes.

DÉFINITION 31 (DÉTERMINISME DES COMPORTEMENTS) *Une méthode présente des comportements dits déterministes si elle ne présente qu'une et une seule terminaison possible, en fonction de l'état actuel du système et des valeurs des paramètres de la méthode : soit la terminaison normale, soit l'une des terminaisons exceptionnelles.*

Comme nous l'avons vu au chapitre 4, JML autorise l'écriture de comportements non déterministes, en particulier lorsque ceux-ci se présentent sous la forme suivante :

```
/*@  behavior
  @    requires P;
  @    assignable A;
  @    ensures Q;
  @    signals (Exception) S;
  @*/
Type m(...) { ... }
```

Dans ce cas de figure, à partir d'une précondition P donnée, il est impossible de connaître la terminaison de la méthode : soit la méthode termine normalement en établissant la postcondition Q, soit la méthode termine exceptionnellement en établissant la postcondition S. Néanmoins, il ne s'agit pas pour autant d'une détection purement syntaxique car il est également possible d'avoir des préconditions qui s'intersectent dans des blocs de spécification ayant des terminaisons distinctes. Nous entendons ainsi mettre en place un mécanisme de résolution de contraintes général pour permettre d'obtenir le cas décrit en figure 8.1. Sur cette figure, les préconditions $P_1, \ldots, P_N$ mènent à une terminaison normale, tandis que les préconditions $P_{N+1}, \ldots, P_{N+M}$ mènent à une terminaison exceptionnelle. Nous souhaitons ainsi vérifier que la spécification permet d'exprimer que la méthode termine de façon déterministe par rapport à sa précondition, soit normalement, soit dans un des cas bien identifié de terminaison exceptionnelle.

Préconditions trop fortes

Dans le cadre de la génération de tests, comme nous l'avons vu au chapitre précédent, les données de test sont calculées sous l'hypothèse des invariants du système. Nous souhaitons vérifier que les comportements du modèle sont activables par rapport aux états décrits dans l'invariant. Si ce n'est pas le cas, les préconditions du comportement sont dites "trop fortes".

Illustrons d'abord ce cas de figure sur un exemple, avant de le définir.

Exemple 8.2 (Précondition trop forte) Soient les deux classes suivantes.

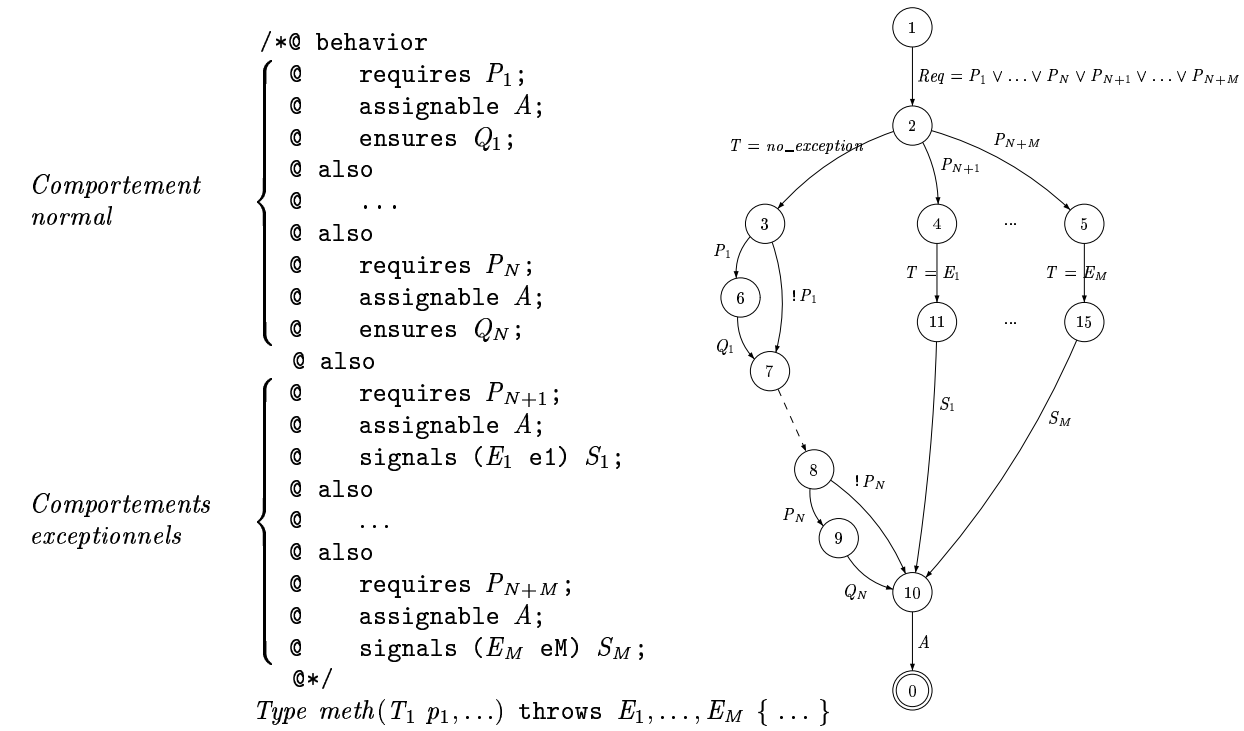


FIG. 8.1 – Illustration des comportements de méthodes déterministes

```
class Test {

 /*@ requires objA.objB == null;
   @ assignable ...
   @ ensures ...
   @*/
 void meth(A objA) { ... }

...
}
```

```
class A {

   B objB;

  //@ invariant objB != null;

}
```

Il apparaît alors une inconsistance entre l'invariant de la classe `A` et la précondition de la méthode `meth(A)` portant sur l'attribut `objB` du paramètre.

Une méthode ayant une précondition trop forte empêche la méthode d'être activée. Cette propriété est déjà détectée en testant la satisfaisabilité des invariants car elle fait échouer cette vérification. Néanmoins, elle permet d'affiner ce résultat en détectant que l'erreur provient de l'impossibilité d'activer le comportement, et non pas d'une incohérence entre l'état après et l'invariant.

DÉFINITION 32 (PRÉCONDITIONS TROP FORTES) *Nous cherchons à vérifier qu'il existe au moins un état du système qui permet d'activer un comportement Cpt, conformément aux invariants.*

$$\exists\, s, \rho \;.\; s \in \mathcal{S}_o \bigwedge_{c \in AllC} Inv_C(s) \;\wedge\; Cpt_{avant}(\rho)(s) \tag{8.29}$$

Ici, il ne s'agit pas d'une erreur de modélisation en soi, mais un comportement inactivable est à détecter dans un processus de génération de tests, puisqu'un tel comportement ne peut pas représenter une cible de test, ni être utilisé dans le calcul du préambule lors de la construction du cas de test.

Préconditions/postconditions trop faibles

Pour finir, nous cherchons à détecter des préconditions ou des postconditions trop faibles. Celles-ci correspondent à des valeurs décrites dans la spécification de méthode comme autorisant l'activation d'une méthode, établissant un postcondition trop faible qui introduit des valeurs d'attributs, supposés modifiés, mais non assignés à l'état d'après.

```
/*@ requires x >= 0;
  @ assignable y;
  @ ensures x < 100 ==> y == x;
  @ ensures x >= 100 ==> (y >= 0 && y < x);
  @*/
 void meth(short x)  ...
```

Dans le cas de cette spécification de méthode, on remarque que seules les valeurs potentielles du paramètre `x` qui sont positives et inférieures à 100 permettent de définir une valeur pour l'attribut `y` supposé être modifié. Nous souhaitons donc informer le spécifieur qu'il existe certaines valeurs des paramètres qui ne sont pas forcément utilisées et qui créent une ambiguïté dans la spécification. Dans l'exemple ci-dessus, la valeur après de `y` n'est pas précisée lorsque `x >= 100`. Libre alors à lui de considérer cet avertissement ou non pour améliorer son modèle. Pour ce faire, il peut soit renforcer la précondition du modèle, soit renforcer la postcondition pour définir la valeur de `y` lorsque `x >= 100`.

Après avoir défini les propriétés que nous souhaitons voir vérifiées sur les modèles que nous considérons, à la fois en terme de cohérence du modèle et de validité de celui-ci, nous présentons les techniques de détections employées. Ces techniques s'appuient sur la résolution de contraintes et l'animation symbolique du modèle.

8.3 Détection des propriétés

Nous décrivons ici les détections de propriétés et les résultats que nous entendons obtenir. Ces résultats se caractérisent par la détection d'une propriété non vérifiée sur le modèle, et le calcul, par instanciation d'un système de contraintes, d'un contre-exemple illustrant la faiblesse du modèle.

Certaines vérifications sont réalisées de manière statiques en mettant en jeu la résolution de contraintes. D'autres sont basées sur l'animation symbolique du modèle pour détecter des erreurs. A chaque détection est associée un contre-exemple. Si la détection est statique, le contre-exemple produit un état (ou un couple d'états avant-après) invalidant la propriété. Si la détection est dynamique, alors le contre-exemple correspond à une trace d'exécution qui mène de l'état initial à un état d'erreur.

Nous prenons ces deux catégories pour classer les détections que nous proposons. Nous illustrons les cas les plus délicats avec un exemple de résultats obtenus.

8.3.1 Des états symboliques aux systèmes de contraintes

Notre approche considère les états symboliques comme un système de contraintes portant sur les variables d'état du système. Ainsi, nous associons à un état symbolique une représentation sous la forme d'un couple $\langle C_V, Env_V \rangle$, composé d'un environnement Env_V, liant les variables du store aux variables du modèle, et un système de contraintes C_V qui portent sur les variables V du store. Ainsi, nous avons :

$$\mathcal{S}_o \equiv \langle I_{Typ}(V), Env_V \rangle \tag{8.30}$$

qui désigne l'état symbolique représentant l'ensemble des états objets du système $\mathcal{S}_o$.

Ainsi, dans le cadre général, les états symboliques $\langle C_V, Env_V \rangle$ sont supposés bien structurés :

$$C_V \Rightarrow I_{Typ}(V) \tag{8.31}$$

La vérification d'une propriété P par rapport à un état symbolique est réalisée en considérant la propriété P et le système de contraintes C_V. Nous introduisons ici la notation $P(V)$ qui désigne l'expression du prédicat P par rapport aux variables de V. Le tableau 8.2 récapitule en fonction du système de contraintes à résoudre, les différentes conclusions sur la satisfaisabilité, l'insatisfaisabilité, la validité et la non-validité d'une propriété P.

Système de contraintes à considérer	Consistance	Conclusion sur P
$C_V \wedge P(V)$	consistant	satisfaisable
	inconsistant	insatisfaisable
$C_V \wedge \neg P(V)$	consistant	invalide
	inconsistant	valide

TAB. 8.2 – Conclusions possibles sur une propriété

Dans les cas où le système de contraintes devient inconsistant suite à la pose des contraintes de $P(V)$, une instanciation du système de contraintes (avant la pose de ces contraintes) donne un contre-exemple à la satisfaisabilité de la propriété. De manière similaire, par réfutation, dans les cas où le système de contraintes reste consistant suite à la pose des contraintes de $\neg P(V)$, le système résultant présente un contre-exemple à la validité de la propriété.

Ces principes nous permettent d'exprimer les propriétés que nous souhaitons détecter en terme de résolution de systèmes de contraintes.

Ainsi, en considérant un état symbolique $\langle C_V, Env_V \rangle$, un comportement *cpt* est applicable si et seulement si :

$$C_V \ \wedge Cpt_{avant}(V \cup \rho) \ \text{ est consistant} \tag{8.32}$$

où $Cpt_{avant}(V \cup \rho)$ représente l'expression de la précondition du comportement cpt par rapport aux variables d'états V et des paramètres ρ.

Soit $\langle C_V, Env_V \rangle$ la représentation d'un état symbolique s_{c_1} par un système de contraintes, et soit $\langle C'_{V'}, Env_{V'} \rangle$ la représentation d'un état symbolique s_{c_2} par un autre système de contraintes. La relation suivante unit ces deux systèmes de contraintes :

$$C'_{V'} = C_V \wedge Cpt_{avant}(V \cup \rho) \wedge Cpt_{apres}(V \cup V' \cup \rho) \tag{8.33}$$

où $Cpt_{avant}(V \cup \rho)$ (resp. $Cpt_{apres}(V \cup V' \cup \rho)$) désigne l'expression du prédicat Cpt_{avant} (resp. Cpt_{apres}) par rapport aux variables de V (resp. aux variables de V et V') et aux paramètres de ρ.

8.3.2 Détections statiques

Les détections statiques utilisent la résolution de contraintes pour détecter des propriétés sur les modèles Java/JML. Nous expliquons dans cette partie comment procéder pour détecter les propriétés précédemment définies. Ces détections utilisent le principe des représentation symbolique des états Java par des systèmes de contraintes, tels que définis dans le chapitre 5.

Vérification des contraintes initiales

Nous cherchons à vérifier les contraintes initiales de la spécification, en nous assurant qu'à chaque création d'objet, les contraintes initiales associées au type de l'objet sont bien satisfaisables ou valides suivant le type de vérification considérée.

Pour vérifier la satisfaisabilité des contraintes initiales, nous considérons la résolution du système de contraintes suivant, pour chaque comportement cpt^{new} extraits à partir des constructeurs de la classe C considérée, créant l'objet o :

$$\begin{aligned} I_{Typ}(V) \ \wedge \ I_{Typ}(V') \ \wedge \ (\exists\, \rho \,.\, & Cpt^{new}_{avant}(V \cup \rho) \wedge \\ & Cpt^{new}_{apres}(V \cup V' \cup \rho)) \ \wedge \ Initially_C(V'[this := o]) \end{aligned} \tag{8.34}$$

où $Initially_C(V[this := o])$ représente les contraintes associées aux contraintes initiales pour lesquelles les références syntaxiques à l'objet $this$ sont substituées par des références à l'objet o nouvellement créé.

Si ce système de contraintes est consistant, alors les contraintes initiales sont satisfaites par l'exécution du constructeur considéré.

Pour vérifier la validité des contraintes initiales, nous considérons la résolution du système de contraintes suivant, pour chaque comportement cpt_{new} extrait à partir des constructeurs de la classe C considérée créant l'objet o :

$$\begin{aligned} I_{Typ}(V) \ \wedge \ I_{Typ}(V') \ \wedge \ (\exists\, \rho . & Cpt^{new}_{avant}(V \cup \rho) \wedge \\ & Cpt^{new}_{apres}(V \cup V' \cup \rho)) \ \wedge \ \neg Initially_C(V'[this := o]) \end{aligned} \tag{8.35}$$

Si ce système de contraintes est consistant et, par conséquent, admet une solution, cette dernière représente un contre-exemple à la validité des contraintes initiales décrites dans le modèle.

Consistance de l'invariant

La consistance de l'invariant se vérifie en ajoutant au système de contraintes représentant les états objets, les contraintes relatives à l'invariant *pour chaque* instance de classe.

Soit $AllC$ l'ensemble des classes considérées et soit $Inv_C(o)$ le prédicat représentant l'invariant de la classe C, appliqué à l'objet o de type C. Le système de contraintes nous permettant de détecter la consistance de l'invariant SC_{inv} est défini comme suit :

$$SC_{inv} = I_{Typ}(V) \wedge \bigwedge_{c \in AllC} I_C \neq \emptyset \wedge \forall\, o.o \in I_C \Rightarrow Inv_C(o)(V) \tag{8.36}$$

Nous cherchons par la contrainte $I_C \neq \emptyset$ à éviter une solution triviale qui consiste à n'avoir aucun objet créé et qui permet de satisfaire (plus aisément) l'invariant. Nous cherchons ensuite à nous assurer que SC_{inv} est consistant en demandant une solution au moteur de résolution de contraintes.

Si le système n'admet pas de solution, alors il existe une inconsistance entre les invariants des différentes classes.

Exemple 8.3 (Consistance de l'invariant) Illustrons cette détection sur un exemple. Considérons une classe `test` qui utilise la classe `Purse`, issue de notre exemple fil rouge, à travers un attribut sur lequel est posé un invariant.

```
class Test

   //@ invariant p != null && p.getBalance() < 0;
   Purse p;

   /*@ requires p1 != null;
     @ assignable p;
     @ ensures p == p1;
     @*/
   public Test(Purse p1) { ... }
```

Cet invariant spécifie que l'attribut `p` doit être différent de `null` et que la valeur associée au champ `balance` doit être strictement négative. Le système de contraintes associé est le suivant (pour plus de lisibilité, seuls les éléments pertinents sont conservés) :

$$I_{Purse} \neq \emptyset \wedge I_{Test} \neq \emptyset \wedge P \in I_{Test} \rightarrow (I_{Purse} \cup \{null\}) \wedge B \in I_{Purse} \rightarrow -32768..32767 \wedge$$
$$(\forall\, o.o \in I_{Test} \Rightarrow P(o) \neq null \wedge B(P(o)) < 0) \wedge (\forall\, o.o \in I_{Purse} \Rightarrow B(o) \geq 0)$$

où P est la fonction représentant l'attribut `p` de la classe `Test` et B est la fonction représentant l'attribut `balance` de la classe `Purse`.

On rappelle que l'invariant de la classe `Purse` spécifie que l'attribut `balance` est toujours positif ou nul. Le moteur de résolution de contraintes ne trouve aucune solution à ce système, ce qui nous permet de déduire que le système, et par conséquent l'invariant, est inconsistant.

Vérification de l'invariant

Nous traitons ici de la vérification de l'invariant dans le cadre de JML. Nous partons du principe que l'invariant exprime des propriétés que l'utilisateur souhaite voir vérifiées sur son modèle, et non des propriétés qui sont supposées renforcer le code qui n'est pas considéré dans notre approche.

La préservation de l'invariant suite à l'exécution d'une méthode doit établir que si la méthode est appelée à partir d'un état respectant l'invariant, alors l'état atteint doit satisfaire lui aussi l'invariant.

Pour vérifier la satisfaisabilité de l'invariant, nous considérons la résolution du système de contraintes suivant, pour chaque comportement *cpt* :

$$\begin{array}{l} I_{Typ}(V) \ \wedge \ \bigwedge_{C \in AllC} \forall\, o.o \in I_C \Rightarrow Inv_C(V[this := o])) \ \wedge \\ \\ I_{Typ}(V') \ \wedge \ (\exists\, \rho \ .\ Cpt_{avant}(V \cup \rho) \ \wedge \ Cpt_{apres}(V \cup V' \cup \rho)) \ \wedge \\ \\ \bigwedge_{C \in AllC}(\forall\, o.o \in I'_C \ \wedge \ Inv_C(V'[this := o])) \end{array} \tag{8.37}$$

Si une solution est trouvée, alors l'invariant est satisfaisable après l'exécution du comportement.

Pour vérifier la validité de l'invariant, nous considérons la résolution du système de contraintes suivant, pour chaque comportement *cpt* :

$$\begin{array}{l} I_{Typ}(V) \ \wedge \ \bigwedge_{C \in AllC} \forall\, o.o \in I_C \Rightarrow Inv_C(V[this := o])) \ \wedge \\ \\ I_{Typ}(V') \ \wedge \ (\exists\, \rho \ .\ Cpt_{avant}(V \cup \rho) \ \wedge \ Cpt_{apres}(V \cup V' \cup \rho)) \ \wedge \\ \\ \bigvee_{C \in AllC} \exists\, o \ .\ o \in I'_C \ \wedge \ \neg Inv_C(V'[this := o])) \end{array} \tag{8.38}$$

Si une solution est trouvée, elle présente un contre-exemple à la préservation de l'invariant d'un état à l'autre.

Exemple 8.4 (Vérification de l'invariant suite à l'exécution d'une méthode) Nous considérons l'exemple donné en partie 4.4.2 dans le chapitre 4. Nous nous intéressons à vérifier l'invariant dans le cas de la méthode `credit(short)`.

Le système de contraintes à résoudre est donc le suivant (pour plus de clarté, seuls les éléments les plus pertinents ont été conservés) :

$$\begin{array}{l} I_{Purse} \subseteq H \ \wedge \ B \in I_{Purse} \rightarrow -32768..32767 \wedge (\forall\, o.o \in I_{Purse}.B(o) \geq 0) \ \wedge \\ this \in I_{Purse} \ \wedge \ A \in -32768..32767 \ \wedge \ A > 0 \ \wedge \\ B'(this) = (((B(this) \ + \ 32767) \ mod \ 65535) \ - \ 32767) \ \wedge \ \exists\, o.o \in I_{Purse}.B'(o) < 0) \end{array} \tag{8.39}$$

Dans cette équation, B est la fonction représentant le champ `balance` de la classe `Purse`, A est la valeur du paramètre de la méthode `credit(short)`, et *this* est l'instance sur laquelle s'applique la méthode. Ce système de contraintes est déduit comme consistant par CLPS-BZ qui instancie la solution suivante :

$$I_{Purse} = \{a_0\}, B = \{a_0 \mapsto 32767\}, this = a_0, A = 1, B' = \{a_0 \mapsto -32768\} \tag{8.40}$$

Ce contre-exemple met en évidence une erreur classique de dépassement de capacité de stockage d'une donnée dans un entier `short`. Nous reviendrons sur cet exemple ultérieurement pour présenter le calcul de trace menant au contre-exemple.

Consistance des contraintes historiques

La consistance des contraintes historiques vise à s'assurer que deux états peuvent être liés par des contraintes historiques. Il s'agit ainsi de trouver un couple d'états tel que la contrainte historique soit satisfaite.

Pour ce faire, nous considérons le système de contraintes suivant pour chaque contrainte historique HC_C de chaque classe C :

$$\exists\, o.o \in I_C \ \wedge\ I_{Typ}(V) \ \wedge\ I_{Typ}(V') \ \wedge\ HC_C(V \cup V')(o) \tag{8.41}$$

pour lequel nous recherchons une solution. Si aucune solution ne peut être trouvée, alors la contrainte est déclarée comme inconsistante.

Nous cherchons ensuite à nous assurer que les comportements satisfont les contraintes historiques. Ainsi, pour vérifier la satisfaisabilité d'une contrainte historique, pour chaque comportement cpt extrait d'une spécification de méthode d'une classe C, et appliqué à un objet o, nous résolvons le système de contraintes suivant :

$$\begin{array}{c} I_{Typ}(V) \ \wedge\ I_{Typ}(V') \wedge Cpt_{avant}(V \cup \rho[this := o]) \wedge \\ Cpt_{apres}(V \cup V' \cup \rho[this := o]) \ \wedge\ HC_C(V \cup V' \cup \rho[this := o]) \end{array} \tag{8.42}$$

Si ce système est consistant, nous concluons sur la satisfaisabilité des contraintes historiques entre deux états.

Pour vérifier la satisfaisabilité d'une contrainte historique, pour chaque comportement cpt extrait d'une spécification de méthode d'une classe C, et appliqué à un objet o, nous résolvons le système de contraintes suivant :

$$\begin{array}{c} I_{Typ}(V) \ \wedge\ I_{Typ}(V') \ \wedge\ Cpt_{avant}(V \cup \rho[this := o]) \wedge \\ Cpt_{apres}(V \cup V' \cup \rho[this := o]) \wedge \neg HC_C(V \cup V' \cup \rho[this := o]) \end{array} \tag{8.43}$$

Si ce système admet des solutions, ces solutions présentent des contre-exemples à la validité des contraintes historiques entre deux états d'exécution.

Déterminisme des comportements

Le déterminisme des comportements se détecte en considérant les différentes préconditions, et les différentes terminaisons.

L'objectif est de rechercher un état du système qui satisfait à la fois les préconditions d'un comportement normal et les préconditions d'un comportement exceptionnel. Il en sera de même pour les comportements exceptionnels entre eux.

Pour détecter cette propriété, nous considérons tous les couples (cpt^{norm}, cpt^{exc}) où cpt^{norm} est un comportement terminant normalement, et cpt^{exc} est un comportement qui

terminera en déclenchant une exception. Pour chacun de ces couples, nous considérons le système de contraintes suivant :

$$I_{Typ}(V) \ \wedge \ \bigwedge_{C \in AllC}(Inv_C(V)) \wedge \\ (\exists\, \rho \ .\ Cpt^{norm}_{avant}(V \cup \rho[this := o]) \wedge Cpt^{exc}_{avant}(V \cup \rho[this := o])) \tag{8.44}$$

Si ce système de contraintes est consistant, un solution présente un contre-exemple illustrant un non-déterminisme entre un comportement normal et un comportement exceptionnel.

Pour être complet, nous considérons également tous les couples de comportements exceptionnels (cpt^{exc1}, cpt^{exc2}), déclenchant des exceptions différentes, et qui doivent par conséquent être déterministes. Nous résolvons de la même manière le système de contraintes suivant :

$$I_{Typ}(V) \ \wedge \ \bigwedge_{C \in AllC}(Inv_C(V)) \wedge \\ (\exists\, \rho \ .\ Cpt^{exc1}_{avant}(V \cup \rho[this := o]) \wedge Cpt^{exc2}_{avant}(V \cup \rho[this := o])) \tag{8.45}$$

Si ce système de contraintes est consistant, un solution présente un contre-exemple illustrant un non-déterminisme entre deux comportements exceptionnels.

Préconditions trop fortes

Nous proposons de détecter l'activation des comportements en détectant les préconditions qui sont trop fortes par rapport à l'invariant. Nous résolvons ainsi le système de contraintes suivant, pour chaque comportement $cpt(\rho)$:

$$I_{Typ}(V) \ \wedge \ \bigwedge_{C \in AllC} (Inv_C(V[this := o])) \ \wedge \exists\, \rho \ .\ Cpt_{avant}(V \cup \rho[this := o]) \tag{8.46}$$

Si ce système de contraintes ne présente pas de solutions, ceci signifie que les préconditions du comportement sont trop fortes.

Préconditions/postconditions trop faibles

Cette dernière détection peut être réalisée de manière syntaxique, en vérifiant si le comportement renseigne bien toutes les nouvelles valeurs des attributs décrits comme `assignable`.

Ainsi, il est impératif que ces attributs soient utilisés dans la postcondition sous la forme `att ==` ou `== att`. L'absence de ces renseignements empêche l'exécution de la spécification et ainsi l'animation ne peut s'opérer de manière déterministe.

8.3.3 Calcul de traces de contre-exemple

Nous décrivons ici les détections que nous pouvons effectuer en mettant en place un mécanisme d'animation symbolique. Ceci nous permet de produire des séquences d'exécutions symboliques qui fournissent une trace de contre-exemple, lorsqu'une propriété n'a pas été vérifiée.

Contre-exemple à la validité de l'invariant

Lorsque l'invariant est détecté comme non valide, il peut être intéressant de chercher un contre-exemple permettant d'atteindre, depuis l'état initial, un état amenant à la violation de l'invariant.

Pour ce faire, nous isolons la partie de l'invariant qui a été mise en défaut et nous lançons une recherche à l'aide d'un algorithme de type *meilleur d'abord* (Best-First) qui vise à atteindre, pour une certaine profondeur donnée, un état satisfaisant la négation de l'invariant. Ceci nous permet de construire un contre-exemple complet qui peut être utilisé par l'utilisateur pour renforcer son modèle. Néanmoins, pour garantir la terminaison du calcul de contre-exemple, nous introduisons une borne limitant la profondeur de recherche. Ainsi, si le calcul de contre-exemple ne trouve pas de solution, cela ne signifie pas pour autant qu'un contre-exemple n'est pas atteignable.

Exemple 8.5 (Contre-exemple menant à la violation de l'invariant) Considérons l'exemple donné au chapitre 4, en partie 4.4.2. La vérification de l'invariant a échoué (exemple 8.4) et a permis de produire un contre-exemple désigné par le système de contraintes (instancié) suivant :

$$I_{Purse} = \{a_0\}, B = \{a_0 \mapsto 32767\}, B' = \{a_0 \mapsto -32768\} \tag{8.47}$$

En prenant comme cible la négation de l'invariant considéré, soit `balance < 0`, l'algorithme de recherche permet d'exhiber le contre-exemple suivant (pour plus de lisibilité, ce contre-exemple a été réifié en notation Java) :

```
Purse p = new Purse();
p.credit(32767);
p.credit(1);
```

qui nous donne une trace menant à la violation de l'invariant.

Complément à la preuve

Nous proposons dans cette sous-partie une application du calcul de contre-exemple dans le cadre du complément à la vérification de propriétés par preuve.

La preuve est un mécanisme de déduction qui a l'avantage de permettre de conclure sur la validité ou l'invalidité de formules mettant en jeu des structures de données aussi bien finies qu'infinies. En comparaison, l'approche que nous avons présentée utilisant la résolution de contraintes ne peut fonctionner qu'avec des domaines finis. Néanmoins, il est assez fréquent que la preuve de théorèmes diverge, i.e., qu'elle ne puisse pas décider la valeur de vérité d'une formule d'obligation de preuve. Nous proposons donc d'utiliser l'animation en complément à la preuve pour permettre d'aider à donner un verdict sur les obligations de preuve non prouvées, ou pour lesquelles la preuve n'a pas pu conclure.

Ainsi, nous proposons le diagramme de collaboration décrit en figure 8.2. Nous partons d'une spécification paramétrée. Dans le cadre de Java/JML, le paramètre est introduit au

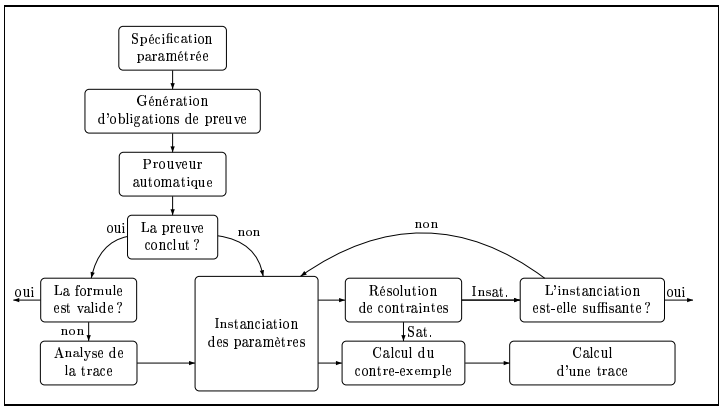


FIG. 8.2 – Schéma de collaboration preuve/résolution de contraintes

niveau de l'ensemble des adresses mémoires, fixée dans notre approche de résolution de contraintes, mais qui dans l'absolu est de taille inconnue, et donc peut-être vu comme un paramètre. Cette spécification produit des obligations de preuve qui sont ensuite déchargées dans un prouveur de théorèmes (comme par exemple haRV^{ey} [DR03]). Si le prouveur conclut, soit la formule est valide et la spécification est prouvée, soit la formule est invalide et l'analyse de la trace du prouveur nous permet d'obtenir un contre-exemple sous la forme d'une conjonction de littéraux ayant causé l'échec de la preuve. La résolution de contraintes prend alors le relais pour tenter de construire un contre-exemple. Néanmoins, une première étape d'instanciation des paramètres est effectuée dans l'objectif de produire une spécification utilisable en résolution de contraintes. La trace du prouveur est ensuite utilisée pour présenter un contre-exemple à l'utilisateur. Le calcul d'une séquence d'exécution menant à un état contre-exemple peut ensuite être mis en œuvre. Si la preuve ne parvient pas à fournir un verdict, la spécification est alors instanciée pour permettre d'utiliser la résolution de contraintes, dans l'objectif (*i*) de conclure sur la validité de l'obligation de preuve (par résolution de contraintes sur des domaines finis) et, (*ii*) le cas échéant, de calculer une séquence d'animation menant au contre-exemple.

8.4 Synthèse

Ce chapitre a présenté l'application de l'évaluation symbolique comme une aide à la mise au point de modèles formels utilisables dans notre processus de génération de tests. Après avoir décrit les propriétés que nous souhaitions détecter, à la fois en terme de propriétés sur les états et de propriétés sur les transitions, nous avons décrit la manière de procéder à ces détections.

La motivation principale de ces détections est d'améliorer les modèles utilisés dans l'approche BZ-TESTING-TOOLS permettant la génération de tests à partir de modèles. Les modèles ainsi produits peuvent être analysés pour aider à leur renforcement. Cette détection est automatisée et permet d'exhiber des contre-exemples de manière à aider à l'amélioration du modèle. Ce travail étant basé sur l'exécution symbolique de spécifications BZP, il ne se limite pas aux spécifications JML et peut ainsi être appliqué à toutes les

notations prises en compte dans l'environnement BZ-Testing-Tools.

L'autre motivation est de proposer une approche originale à la vérification de cohérence d'un modèle Java/JML, en vérifiant les propriétés du modèle par rapport au modèle en lui-même, et non par rapport au code. Dans ce cadre, nous avons proposé une approche permettant l'utilisation de la résolution de contraintes et de l'animation symbolique dans le cadre de la vérification de modèle, en complément à l'approche par preuve. La vérification de la cohérence d'un modèle JML en lui-même semble ne jamais avoir été ciblée dans aucune approche. En effet, les approches de preuve impliquant JML sont tournées vers la preuve de conformité du code Java par rapport à la spécification JML. C'est notamment le cas de ESC/Java2 [CK04], Krakatoa [MPMU03], Jack [BRL03] ou encore du projet LOOP [vdBJ01].

Les démarches présentées visent à aider l'utilisateur à établir et à renforcer sa spécification dans l'objectif de s'appuyer sur son modèle pour générer des tests. Ces travaux ont été publiés dans [BDG05] où est décrite la vérification de cohérence d'une spécification JML en passant par le formalisme des machines abstraites B, et dans [BDL05a] où sont définies les bases de vérification d'un certain nombre de propriétés sur les modèles B utilisés pour la génération de tests dans le cadre du projet BZ-Testing-Tools. Les travaux présentés dans ce chapitre sont inspirés de ces résultats et adaptés à JML.

Ce chapitre clôture la description des principes utilisés dans cette thèse. Nous présentons, dans la partie suivante, les réalisations implantant ces principes, et les expérimentations menées sur une étude de cas réaliste issue du milieu industriel.

Troisième partie

Réalisations et expérimentations

Chapitre 9

Le prototype JML-Testing-Tools

Sommaire

Ce chapitre présente le prototype réalisant l'implantation des techniques décrites dans les chapitres précédents. Ce prototype, à vocation académique, est nommé JML-Testing-Tools, en hommage à BZ-Testing-Tools. Ce logiciel se décompose en deux parties : une partie proposant la validation d'un modèle Java/JML, par animation symbolique, et une partie proposant la génération automatique de cas de test fonctionnels pour un programme Java à partir d'un modèle JML de celui-ci.

Le plan de ce chapitre est le suivant. Nous verrons dans une première partie les généralités du prototype JML-Testing-Tools, avant de décrire les deux modules principaux qui sont l'animateur symbolique et le générateur de tests, qui feront l'objet des deux parties suivantes.

9.1 Généralités

Nous présentons dans cette partie les généralités sur le prototype JML-Testing-Tools. Nous commençons par voir l'architecture logicielle globale de l'outil, puis nous

nous intéressons à la partie la plus en amont, le compilateur.

9.1.1 Architecture logicielle

La figure 9.1 présente l'architecture du prototype JML-TESTING-TOOLS. Un fichier Java/JML donné en entrée de l'outil est analysé pour déterminer les références manquantes, c'est-à-dire les classes utilisées pour typer les attributs ou les paramètres. Une fois toutes les classes nécessaires collectées, la structure des classes est stockée dans une structure mémoire nommée Internal Data Structure (IDS).

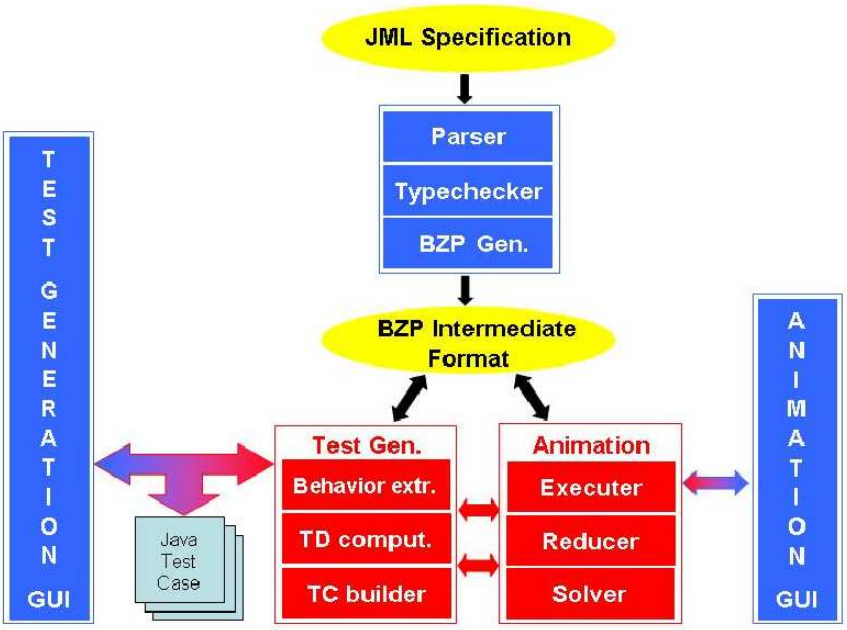


FIG. 9.1 – Architecture du prototype JML-TESTING-TOOLS

Cette structure autorise d'être traversée par des visiteurs [GHJV94]. Un visiteur est en charge de contrôler le type des expressions, ainsi que la correction syntaxique des expressions Java/JML employées dans les clauses de spécifications. Si le contrôle de type ne produit aucune erreur, un second visiteur parcours alors l'IDS pour générer un fichier BZP selon les règles de traduction décrites au chapitre 6.

Le prototype se décompose alors en deux parties : l'animateur et le générateur de tests. Si l'animation est indépendante du générateur de tests, le générateur de tests s'appuie sur la résolution de contraintes et l'animation pour le calcul des objectifs de test (modules Behavior extr. et TD computation) et le calcul des séquences de test abstraites (module TC builder). L'animateur est lui composé des modules usuels de BZ-TESTING-TOOLS (Solver, Reducer et Executer), modifiés suivant les principes décrits en partie 6.4

Nous nous intéressons à présent à la partie commune aux deux modules, le compilateur traduisant un modèle JML en un fichier de format intermédiaire BZP.

9.1.2 Le compilateur JML → BZP

Le compilateur JML est la partie commune aux modules d'animation et de génération de tests. Il se décompose en trois parties distinctes :
- un analyseur syntaxique construisant une instance d'IDS ;
- un contrôleur de type ;
- un générateur de code BZP.

Intéressons-nous à la structure de données interne (IDS). L'IDS est une implantation Java du méta-modèle Java/JML. On désigne par méta-modèle un modèle représentant un modèle. Ainsi le modèle de données de l'IDS représente la structure de classes Java, i.e. les attributs, les méthodes et leurs paramètres, etc. Les éléments représentés par le méta-modèle Java/JML sont donnés en figure 9.2.

On retrouve dans cette représentation les éléments du modèle de données Java : les classes, les attributs de classes, les méthodes et leurs paramètres. Les classes permettent de définir des types de données, associés aux attributs, aux paramètres ou aux valeurs de retour des méthodes. On distingue deux types d'opérations : les méthodes et les constructeurs.

Au modèle de données Java s'ajoute la représentation des clauses de spécification JML. Ces clauses sont regroupées dans des spécifications de méthodes. Chacune des clauses contient un prédicat Java/JML, stocké sous la forme d'un arbre de syntaxe abstraite.

L'intérêt de cette structure est double. Premièrement, il s'agit d'une structure arborescente qui accepte les visiteurs. Il suffit donc de développer les visiteurs adéquat pour ajouter des fonctionnalités. De ce fait, un visiteur de contrôleur de type et un visiteur de génération de code BZP sont successivement employés pour produire un fichier BZP exploitable par le moteur d'animation. Deuxièmement, cette structure permet de donner accès aux différents éléments composant les classes Java/JML qui font l'objet de l'animation. L'interface d'animation s'appuie sur ce format pour connaître les objets et les méthodes Java/JML disponibles et leurs appellations dans le formalisme BZP.

Nous présentons désormais les deux modules qui se greffent sur le fichier au format BZP ainsi créé. Nous commençons par la description de l'animateur symbolique de modèles JML, puis nous enchaînons avec la présentation du générateur de tests.

9.2 Animateur symbolique

L'animateur JML-Testing-Tools fournit une interface homme-machine permettant à un utilisateur de visualiser les états du système qu'il anime, et lui donnant le possibilité d'actionner les différentes méthodes (créations d'objets, méthodes d'instances, etc.). L'utilisateur peut ainsi voir évoluer les états du système et s'assurer du bon fonctionnement de son modèle.

Détaillons à présent l'interface d'animation.

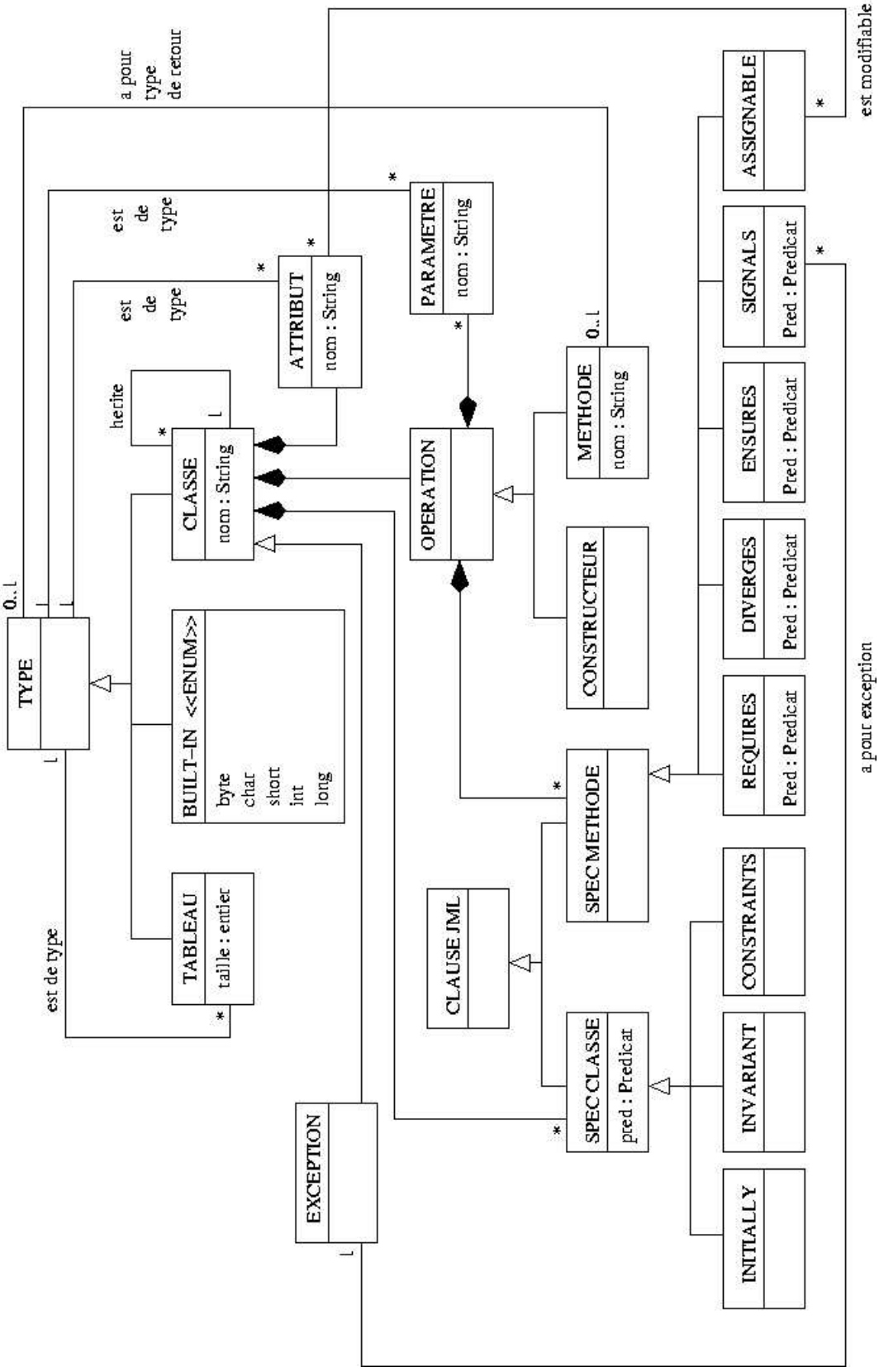


Fig. 9.2 – Méta-modèle Java/JML implanté dans l'IDS

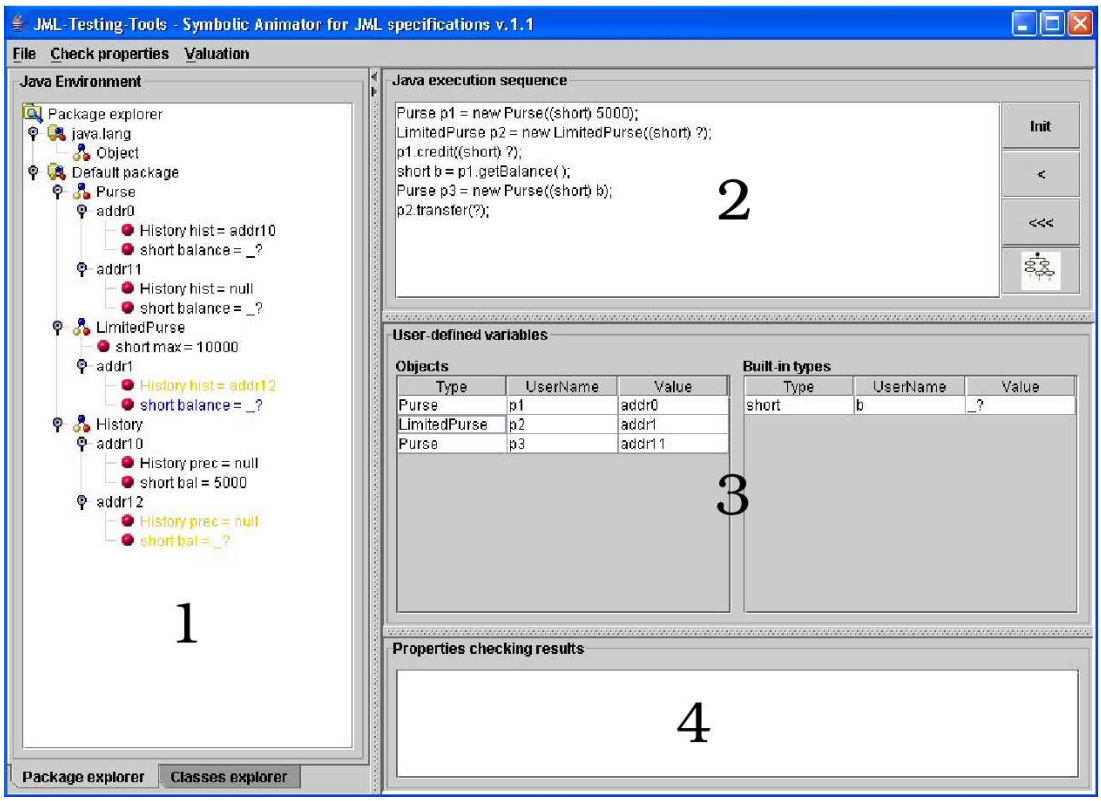


FIG. 9.3 – Interface d'animation de JML-TESTING-TOOLS

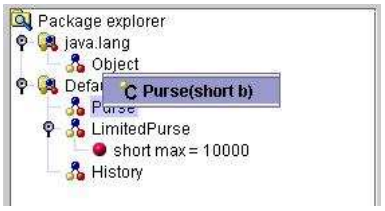
9.2.1 L'interface d'animation

L'interface d'animation est présentée en figure 9.3. Cette interface se décompose en 4 parties, comme représenté sur la capture d'écran. L'utilisateur interagit avec les classes pour créer les différents objets et avec les objets pour invoquer des méthodes. Ces 4 parties sont les suivantes.

1. La partie gauche de l'interface représente l'état du système organisé suivant les packages contenant les classes. Lorsqu'il existe un objet d'une classe, on trouve son adresse, sous la forme $addr0$, $addr1$, etc., à laquelle sont associés les attributs de la classe et leur valeur.
2. La partie en haut à droite donne l'équivalent des actions de l'utilisateur en instructions Java. Les boutons sur la droite de la zone de texte permettent respectivement (de haut en bas) de revenir à l'état initial, de revenir en arrière d'un pas, de revenir en arrière de trois pas, ou d'afficher l'arbre des exécutions décrites par l'utilisateur.
3. La partie au milieu à droite donne les variables locales créées par l'utilisateur. Celles-ci sont réparties en deux catégories : les objets, et les variables de types prédéfinis. Ces variables sont obtenues soit par création d'objets, soit par récupération d'une valeur de retour d'une méthode lors de son invocation.
4. La partie en bas à droite permet d'informer l'utilisateur de la vérification de différentes propriétés (telles que l'invariant ou les contraintes historiques). Si un des

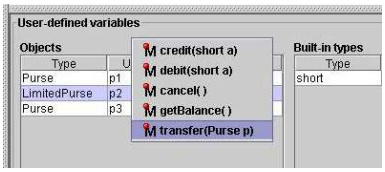
prédicats n'est pas vérifié, cette fenêtre indique lequel et donne la possibilité de voir un contre-exemple.

Une séquence d'animation se déroule de la manière suivante.

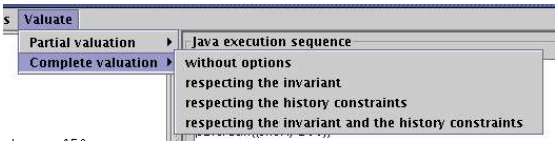


L'utilisateur sélectionne (par un clic du bouton droit) la classe dont il veut créer une instance. Il choisit alors le constructeur qui l'intéresse. Il doit alors saisir un nom pour la variable. Si le constructeur admet des paramètres, il est alors demandé de saisir leurs valeurs. S'il le souhaite, il est possible de laisser un des paramètres non spécifié, de manière à créer un état contraint.

L'objet ainsi créé s'ajoute dans la représentation de l'état et la variable locale associée à l'objet est ajoutée à la liste des variables disponibles (en partie 3 de l'interface).



L'utilisateur peut alors choisir, en cliquant sur cet objet, la méthode à invoquer. Avant de saisir les valeurs des paramètres, l'utilisateur peut choisir le comportement de la méthode qu'il souhaite activer : soit le comportement normal, soit un des comportements exceptionnels décrits dans la spécification de la méthode. Si la précondition de la méthode échoue, l'interface affiche dans la partie 4 les prédicats non satisfaits.



Si un état symbolique a été créé, il est possible pour l'utilisateur d'instancier l'état. Cette instanciation peut être réalisée en ajoutant des contraintes supplémentaires sur l'état symbolique, comme par exemple, satisfaire l'invariant de classe.

9.2.2 Vérification de propriétés lors de l'exécution

JML-Testing-Tools offre la possibilité, lors de l'exécution du modèle, de vérifier pour un état donné les propriétés telles que l'invariant de classe ou les contraintes historiques. Ces propriétés sont vérifiées suivant les principes de résolution de contraintes décrits dans le chapitre 8, permettant ainsi de conclure à l'insatisfaisabilité, la satisfaisabilité, la non-validité ou la validité d'une propriété.

En cas d'échec lors de la vérification, un contre-exemple est instancié, illustrant le défaut à la fois pour l'état dans lequel se trouve l'animation, mais également une trace d'exécution complète arrivant dans cet état. La figure 9.4 illustre la vérification de la validité de l'invariant de l'exemple fil rouge décrit au chapitre 4, suite à la séquence d'exécution suivante :

```
Purse p1 = new Purse(5000);
p1.credit(A);
```

Un contre-exemple est ainsi produit, instanciant la valeur de A à 27768, ce qui amène la valeur du champ `p1.balance` à être négative.

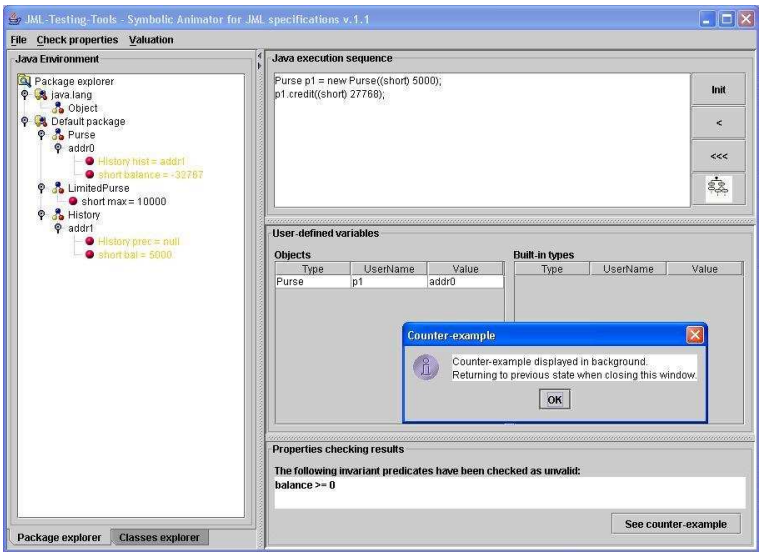


FIG. 9.4 – Vérification de la validité d'un invariant avec JML-TT

9.2.3 Un premier pas vers la génération de tests

En plus des fonctionnalités précédentes, l'animateur JML-TESTING-TOOLS permet d'exporter la séquence d'animation créée par l'utilisateur en un programme Java compilable et exécutable sur les classes considérées pour l'animation. Cette technique est une première étape vers la génération de test. Elle consiste à produire un ou plusieurs cas de test à partir d'un séquence d'animation définie par l'utilisateur.

Si la séquence d'animation à exporter contient des valeurs contraintes, introduites au niveau des paramètres d'une méthode ou d'un constructeur, alors plusieurs fonctions de tests sont produites. Ces différentes fonctions dépendent de l'instanciation des paramètres laissés comme contraints. Pour chaque type de paramètre, nous considérons différents types d'instanciations. Ainsi, les valeurs des paramètres entiers (les entiers ou les caractères) sont prises aux bornes des domaines des variables (en considérant toutefois la somme de toutes les variables entières). Les valeurs des paramètres objets sont énumérées exhaustivement en fonction des instances existantes satisfaisant les contraintes qui s'appliquent sur les paramètres. De ce fait, la valeur `null` peut être sélectionnée si elle n'est pas en contradiction avec les préconditions de comportement de méthode activé.

Illustrons ceci sur un exemple. Considérons la séquence d'exécution suivante, issue de l'exemple JML présenté au chapitre 4 en partie 4.4.2. Supposons que la spécification de la méthode `credit(short)` de la classe `Purse` a été corrigée pour forcer la précondition à n'accepter que des valeurs du paramètre `a` qui n'entraînent pas la violation de l'invariant. La nouvelle précondition est la suivante :

```
/*@ requires a >= 0 && (balance + a) <= 32767;
  @ requires \typeof(this) == \type(Purse); */
```

Nous considérons la séquence d'animation contrainte suivante :

Cas de test	Calcul sur A	Calcul sur B	Calcul sur C
1	minimisation $A = 0$	minimisation $B = 1$	énumération $C =$ `p1`
2	minimisation $A = 0$	minimisation $B = 1$	énumération $C =$ `p2`
3	minimisation $A = 0$	minimisation $B = 1$	énumération $C =$ `p3`
4	maximisation $A = 10000$	maximisation $B = 27767$	énumération $C =$ `p1`
5	maximisation $A = 10000$	maximisation $B = 27767$	énumération $C =$ `p2`
6	maximisation $A = 10000$	maximisation $B = 27767$	énumération $C =$ `p3`

Tab. 9.1 – Exemple de cas de test obtenus à partir d'une séquence d'animation

```
Purse p1 = new Purse(5000);
LimitedPurse p2 = new LimitedPurse(A);
p1.credit(B);
short b = p1.getBalance();
Purse p3 = new Purse(b);
p2.transfer(C);
```

où les paramètres contraints sont signalés par les symboles A, B et C. Le premier paramètre contraint, A, introduit lors de la construction de l'instance `p2`, a pour domaine 0..10000 (entier court positif et inférieur au maximum), le deuxième paramètre contraint B, introduit par l'instruction `p1.credit(`B`)`, a pour domaine l'intervalle 1..27767 (entier court strictement positif et n'entraînant pas de dépassement de capacité). Le troisième paramètre contraint introduit lors de l'instruction `p2.transfer(`C`)` a pour domaine l'ensemble {`p1`, `p2`, `p3`} (instance non-`null` de la classe `Purse`). La génération des tests à partir de cette séquence produira 6 cas de tests, dont le calcul est donné par le tableau 9.1.

Le passage de ces cas de test sur le programme mène à la détection d'une violation de l'invariant de la classe `LimitedPurse` pour le cas de test 4, induit par le transfert d'un porte-monnaie non limité ayant un solde maximal (32767) dans un porte-monnaie limité n'acceptant qu'un solde plafonné à 10000.

Exporter une séquence d'animation est une première étape vers la génération de tests proprement dite. Nous décrivons désormais le fonctionnement du générateur de tests aux limites de JML-Testing-Tools.

9.3 Générateur automatique de tests aux limites

Cette partie décrit la partie génération de tests aux limites du prototype JML-Testing-Tools. Cette interface implante les éléments décrits au chapitre 7. Elle donne à un utilisateur la possibilité de choisir la couverture du modèle, sous la forme des méthodes qu'il souhaite tester, choisir la couverture des disjonctions, en sélectionnant la réécriture des disjonctions, et choisir la couverture des données appliquées aux paramètres. La description de l'interface fera l'objet de la première sous-partie. Les tests générés sont regroupés sous la forme d'une campagne de tests dont nous détaillerons les différents éléments dans une seconde sous-partie.

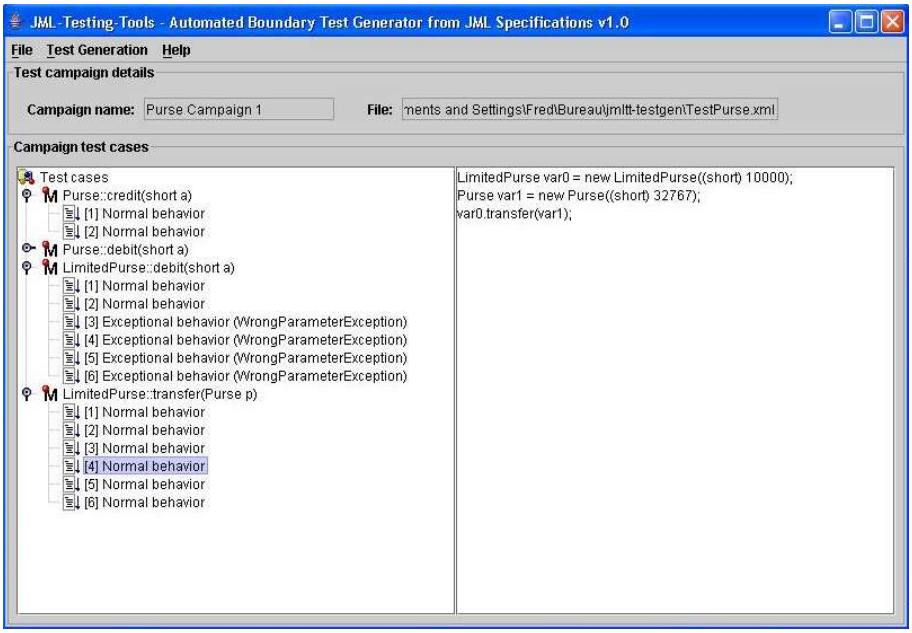


FIG. 9.5 – Fenêtre principale du générateur de tests JML-TESTING-TOOLS

9.3.1 L'interface de génération de tests

L'interface de génération de tests de JML-TESTING-TOOLS fonctionne sur le même principe que l'interface d'animation. Elle commence par charger les différentes classes référencées à partir d'un fichier classe source. L'interface se décompose en trois modules, le module d'extraction des comportements, le module de calcul des valeurs limites, et le module de calcul d'une séquence de test.

La fenêtre principale du générateur de tests est donnée en figure 9.5. Sur cette capture d'écran, nous trouvons :

1. En haut, le descriptif de la séquence de test avec son fichier de sauvegarde associé.
2. A gauche, la liste des cas de test qui ont été produits triés par classe et par méthode. Pour chaque méthode, on trouvera le type de comportement illustré par le cas de test produit.
3. A droite, les instructions composant le cas de test sélectionné dans la liste de gauche.

La sélection des objectifs de test se fait par l'intermédiaire de l'onglet *"Test Target"* donné dans la figure 9.6. Cette fenêtre permet de choisir les méthodes que l'utilisateur souhaite tester.

La sélection des données de tests se fait par l'intermédiaire de l'onglet *"Coverage"* donné par la figure 9.7. Cette fenêtre permet de choisir la réécriture qui s'applique sur les disjonctions et la couverture des données. Ainsi, l'utilisateur sélectionne quelles valeurs numériques doivent être prises en compte (les paramètres numériques et/ou les attributs numériques des paramètres), et quelle couverture leur appliquer (minimisation et/ou maximisation).

Le dernier onglet, *"Test Suite"*, présente les options de génération de tests, notamment la profondeur de recherche de l'algorithme de recherche et le fichier Java de sortie dans

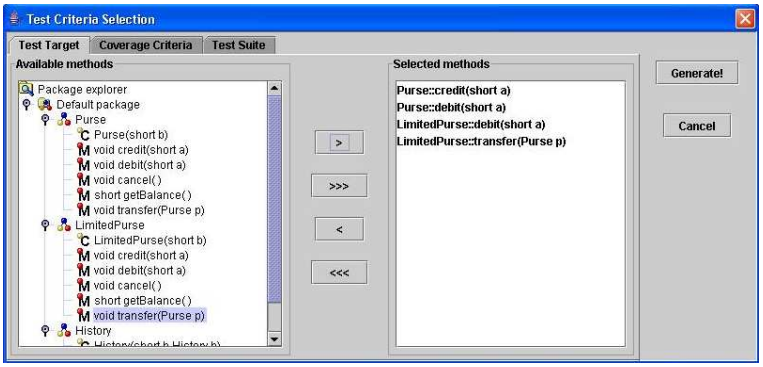


FIG. 9.6 – Choix des objectifs de test dans JML-TESTING-TOOLS

lequel sont écrits les cas de test. La génération des cas de test est encapsulée dans une structure de campagne de tests qui nous décrivons à présent.

9.3.2 Format d'une campagne de tests

Une campagne de tests est composée des éléments décrits sous la forme d'une DTD, donnée en figure 9.8. Ceux-ci permettent de stocker en mémoire les différentes sélections effectuées par l'utilisateur ainsi que les cas de tests en eux-mêmes.

Une *campagne de test* possède un nom et une description. On lui associe une couverture de test, un préambule et des méthodes sous tests. Une *couverture de test* se caractérise par une réécriture, le choix de minimisation et/ou maximisation, et les choix de mise aux limites des paramètres numériques et des attributs des paramètres objets. Le *préambule* se caractérise par une profondeur de recherche de l'algorithme de génération de cas de tests. Les *méthodes sous test* sont caractérisées par une méthode et une classe. On associe à une méthode sous test des *cas de test*. Chacun d'eux se caractérise par un comportement BZP, une exception éventuelle et une suite d'instructions. Les *instructions* concernent une classe, une méthode, une liste de valeurs de paramètres et une variable de retour. Les instructions se décomposent en deux catégories : les *appels de constructeurs* et les *appels de méthodes*. La partie de la DTD correspondant aux instructions a été donnée dans le chapitre 7, en figure 7.5.

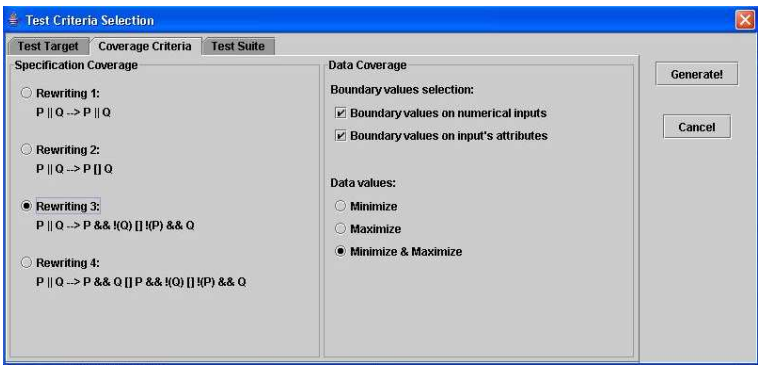


FIG. 9.7 – Choix du critère de couverture dans JML-TESTING-TOOLS

```
<!ELEMENT TestCampaign (Description, Coverage, Preamble, MethodsUnderTest) >
<!ATTLIST TestCampaign
        name CDATA #REQUIRED >
<!ATTLIST TestCampaign
        file CDATA #REQUIRED >

<!ELEMENT Description (#PCDATA) >

<!ELEMENT Coverage EMPTY >
<!ATTLIST Coverage
        rewriting CDATA #REQUIRED
        minimax CDATA #REQUIRED
        bounds_num (true | false) #REQUIRED
        bounds_att (true | false) #REQUIRED >

<!ELEMENT Preamble EMPTY >
<!ATTLIST Preamble
        depth CDATA #REQUIRED >

<!ELEMENT MethodsUnderTest (MethodUnderTest+) >
<!ELEMENT MethodUnderTest (TestCase)+ >
<!ATTLIST MethodUnderTest
        class CDATA #REQUIRED
        method CDATA #REQUIRED >
```

FIG. 9.8 – Schéma XML d'une campagne de tests

9.4 Synthèse

Ce chapitre a présenté une implantation des concepts formulés dans les contributions de ce mémoire. Le prototype JML-TESTING-TOOLS a été développé au sein du LIFC, à l'aide des langages Java (pour le compilateur et les interfaces graphiques), et SICStus Prolog (pour les modules d'animation et de génération de tests). La communication entre ces deux langages est assurée par un mécanisme de *sockets*, envoyant les commandes à l'API Prolog depuis Java, et renvoyant les résultats calculés par Prolog à Java. L'animateur JML-TESTING-TOOLS a été présenté dans le cadre de sessions outils [BDLU05a, BDL06b].

Le chapitre suivant présente l'utilisation de ce prototype sur une étude de cas inspirée du monde industriel.

Chapitre 10

Expérimentations sur une étude de cas

Sommaire

Ce chapitre présente l'application de techniques précédentes à une étude de cas issue du milieu industriel : l'application Demoney développée par Trusted Logics [TL05], et proposée dans le cadre de l'ACI GECCOO. Il s'agit d'une spécification écrite à partir de la spécification (informelle) publique donnée par [MM].

Cette spécification a été écrite pour permettre de se conformer aux structures de données supportées par notre approche : les objets et les entiers. Ainsi, la spécification a pu être validée par l'animateur JML-Testing-Tools. Enfin, une fois la spécification validée, elle peut être utilisée avec le générateur de tests. Les différentes étapes relatant l'application de notre processus à cet exemple sont décrits dans ce chapitre.

Ce chapitre se décompose de la manière suivante. Nous verrons dans un premier temps une description informelle de l'application Demoney. Puis, nous nous intéresserons à la spécification elle-même, qui est validée au fur et à mesure avec l'animateur JML-Testing-

TOOLS. Enfin, nous verrons les objectifs de tests isolés et les cas de tests produits par le générateur de tests JML-TESTING-TOOLS.

10.1 Présentation de l'étude de cas : Demoney

L'application Demoney [MM] est une applet destinée aux Smart Cards, une carte à puce évoluée pouvant embarquer des logiciels écrits en langage JavaCard [Sun00]. Cette applet décrit un porte-monnaie électronique et son fonctionnement à travers les méthodes activables depuis le terminal avec lequel la carte communique. Ces méthodes permettent d'exercer les différentes fonctionnalités de la carte, à savoir personnaliser la carte, créditer de l'argent, ou effectuer un paiement.

Nous décrivons à présent, de manière informelle, les possibilités que doit présenter l'application Demoney.

10.1.1 Présentation générale

Nous présentons ici les principes généraux de l'applet Demoney. Ces principes introductifs s'appliquent sur les méthodes que nous décrirons ultérieurement.

Le déroulement des applications de carte à puce présentent un cycle de vie commençant par la configuration de la carte, suivi de son utilisation. Une fois que la carte est configurée, on dit qu'elle est *personnalisée*, les opérations de transfert, telles que les débits ou les crédits, peuvent s'appliquer.

Toutes les méthodes sont susceptibles d'être invoquées depuis n'importe quel état du système. De ce fait, soit l'invocation a lieu dans un état satisfaisant l'invocation de la méthode et celle-ci termine normalement, soit elle a lieu dans un état non-satisfaisant et la méthode déclenche une exception.

La carte peut être accédée par différents terminaux ayant chacun un niveau d'autorisation spécifique, que nous décrivons plus loin. Le porte-monnaie peut être crédité, moyennant la vérification d'un code PIN dont le nombre de tentatives est limité. Le solde de la carte est conservé dans le porte-monnaie. Il ne peut pas être négatif, mais il est plafonné par un solde maximal. Ainsi, ce plafond limite les montants possibles pour les opérations de crédit. De manière similaire, les opérations de débit sont également plafonnées par un montant maximal autorisé.

Nous présentons à présent les diverses fonctionnalités, en commençant par les informations relatives à la configuration de la carte. Nous verrons ensuite les différents niveaux d'autorisation puis nous terminerons cette présentation par les différentes opérations requises sur la carte.

10.1.2 Les informations de configuration de la carte

La configuration de la carte permet de fixer les éléments suivants :

- le solde maximum contenu sur le porte-monnaie. Ce solde doit être strictement positif et il plafonne le solde courant.
- le montant maximal de débit. Ce montant fixe la valeur maximale qu'il est possible de débiter sur le porte-monnaie, indépendamment du solde courant du porte-monnaie. Ce montant est strictement positif.
- le code PIN. Ce code doit être donné par l'utilisateur pour accéder à certaines opérations du porte-monnaie.
- le nombre d'essais d'identification du code PIN. Cette valeur donne le nombre d'échecs autorisés pour s'identifier avec un code PIN correct.

Ces informations doivent être configurables lors de la phase de personnalisation de la carte. La phase de personnalisation ne peut être effectuée n'importe quand, comme nous l'avons vu précédemment, ni par n'importe qui, comme nous allons le voir à présent.

10.1.3 Les niveaux d'autorisation

On distingue 4 niveaux d'autorisation qui sont associés à 4 types de terminaux. Ceux-ci sont les suivants.

- Le niveau *public* permet de consulter les informations publiques de la carte ; il est associé aux terminaux dits de *consultation*.
- Le niveau *débit* permet d'effectuer un débit, c'est-à-dire, un paiement à partir du porte-monnaie ; il est associé aux terminaux dits de *paiements*, comme ceux que l'on trouve dans les magasins.
- Le niveau *crédit* autorise à créditer le porte-monnaie. Celui-ci est crédité soit à partir d'argent en liquide, soit à partir d'un compte bancaire. Dans ce second cas, l'utilisateur doit fournir un code PIN permettant de faire le retrait sur le compte bancaire (non modélisé ici). Ce niveau est associé à des terminaux *bancaires*.
- Le niveau *administrateur* permet de configurer la carte ; il est associé aux terminaux *d'administration* des banques qui permet de mettre à jour les paramètres de configuration de la carte.

Ces différents niveaux d'autorisations permettent d'activer différentes opérations, que nous présentons maintenant.

10.1.4 Les opérations de Demoney

Les opérations de Demoney sont les suivantes :

- PUT_DATA : permet la mise à jour des informations de configuration. Elle ne peut être appelée qu'au niveau administrateur.
- STORE_DATA : cette opération permet de personnaliser une instance d'une carte. Elle ne peut être appliquée qu'une seule fois, et n'est accessible qu'en mode administrateur.
- INITIALIZE_UPDATE : cette opération permet l'initialisation de l'authentification mutuelle entre le terminal et la carte. Elle permet ainsi de sélectionner le type d'authentification associé à la session en cours. Cette opération doit être suivie de l'invocation de l'opération EXTERNAL_AUTHENTICATE.

- EXTERNAL_AUTHENTICATE : valide l'authentification mutuelle entre le terminal et la carte, et ouvre une session sécurisée.
- PIN_CHANGE_UNBLOCK : cette commande permet de changer la valeur du code PIN ou de réaliser le déblocage de la carte. Celle-ci est bloquée si l'utilisateur a échoué l'identification par code PIN (en rentrant un code PIN erroné successivement de manière à épuiser tous les essais d'identification). Cette méthode n'est accessible qu'en mode administrateur.
- VERIFY_PIN : vérifie le code PIN saisi par l'utilisateur sur un terminal. La vérification ne peut avoir lieu qu'un certain nombre de fois, fixé lors de la configuration de la carte. Le code PIN peut avoir trois états : validé s'il est correct, bloqué s'il est incorrect et qu'il ne reste plus d'essais, non-validé mais non-bloqué s'il est incorrect, mais qu'il reste encore des essais.
- INITIALIZE_TRANSACTION : permet d'initialiser une transaction en vérifiant si celle-ci est correcte vis-à-vis de son type (crédit ou débit) et des autorisations en cours de validité. Cette opération doit forcément être suivie de la commande COMPLETE_TRANSACTION.
- COMPLETE_TRANSACTION : valide la transaction précédemment initiée et vérifiée.

Ces commandes ne peuvent pas toutes être exécutées dans n'importe quel ordre et certains enchaînements d'opérations sont obligatoires : EXTERNAL_AUTHENTICATE suit nécessairement INITIALIZE_UPDATE et INITIALIZE_TRANSACTION est suivi de COMPLETE_TRANSACTION.

Nous décrivons à présent la modélisation réalisée dans le cadre de cet étude de cas.

10.2 Description de la spécification JML

Cette partie décrit la modélisation JML de l'exemple du porte-monnaie Demoney. Celle-ci est réalisée à partir du document de la spécification publique de Demoney disponible dans [MM]. Le mécanisme de la JavaCard a également été adapté pour se conformer aux structures de données supportées. Par choix, nous cherchons à faire une modélisation Java/JML bien que l'exemple soit issu du monde de la JavaCard. Ainsi, notre approche n'étant pas dédiée aux applets JavaCard, qui présentent des différences non négligeables par rapport à Java, nous ne cherchons pas à nous conformer au plus près à un tel programme. L'adaptation complète de l'approche au programmes JavaCard fait l'objet d'une des perspectives de ce travail, comme nous le verrons dans la quatrième partie de ce mémoire.

Néanmoins pour conserver une proximité relative aux applets JavaCard, nous effectuons notre modélisation de manière à ce qu'il soit aisé, par un mécanisme simple d'*adaptation*, de transformer les cas de tests obtenus en cas de tests utilisables sur une implantation destinée à la JavaCard.

Nous commençons par voir les modifications effectuées sur l'exemple pour l'appliquer à notre technologie. Puis nous nous intéresserons aux constantes et aux attributs permettant

de conserver les informations relatives au porte-monnaie.

10.2.1 Adaptation du mécanisme de la JavaCard

Nous présentons ici les adaptations nécessaires relatives au mécanisme de la JavaCard, après avoir introduit le principe de fonctionnement de la JavaCard.

Le mécanisme de la JavaCard

Les programmes JavaCard sont embarqués sur des cartes à puce. Le langage JavaCard s'exécute sur un JavaCard Runtime Environment (JCRE). Lorsqu'une carte est insérée dans un lecteur de carte, ou Card Acceptance Device (CAD), le CAD sélectionne une applet sur la carte et lui envoie un série de commandes à exécuter. Chaque applet est identifiée et sélectionnée par son identifiant d'application (Application IDentifier – AID). Les commandes exécutées sont formatées et transmises sous la forme d'objets APDU (Application Protocol Data Units). Les applets répondent à chaque APDU avec un status word (SW) qui indique le résultat de l'opération.

Le paramètre APDU

La JavaCard utilise une méthode principale nommée "process" qui prend en paramètre un objet de type APDU (Application Protocol Data Unit). Cet objet structuré est composé de 5 bytes obligatoires et 32 bytes facultatifs.

Les 5 bytes obligatoires sont les suivants :
- CLA : byte d'identification de l'applet ;
- INS : byte d'instruction, spécifie quelle instruction doit être exécutée ;
- P1 : byte correspondant à un premier paramètre ;
- P2 : byte correspondant à un second paramètre ;
- Lc : byte donnant la taille du champ de données composant le paramètre, codé sur les bytes facultatifs.

Pour simplifier la modélisation, nous considérons une méthode par instruction possible, qui admet les paramètres représentant les 5 bytes obligatoires et un entier court signalant un paramètre additionnel. La raison est double. Premièrement, le découpage des méthodes représentant les actions possibles permet une plus grande lisibilité du modèle JML. Deuxièmement, nous souhaitons éviter d'avoir à créer systématiquement un objet APDU pour le passage d'un paramètre à une méthode. En effet, la création d'un tel objet pour activer chaque méthode va avoir pour effet (i) de ralentir le calcul de préambule et (ii) de diminuer de moitié le nombre de méthodes contenues dans le préambule – pour N méthodes à appeler, on dénombrera N créations d'objets APDU. Ici encore, le support natif du principe JavaCard par notre approche permettrait de passer outre cette restriction, en considérant une création systématique d'un objet APDU.

Le code PIN

Le code PIN est supposé être modélisé par un objet OwnerPIN, qui précise, entre autres, le nombre d'essais pour la vérification du code PIN, la taille du code PIN, et sa

valeur sous la forme d'une chaîne de caractères d'autant de digits. Dans les bibliothèques JavaCard, cet objet met en place des tableaux, qui ne sont pas supportés par notre approche.

Pour simplifier, nous utiliserons un objet OwnerPIN "customisé" qui permet de spécifier le code PIN par un entier court de 4 chiffres (entre 0000 et 9999), ainsi que le nombre d'essais et les états du code PIN (défini et validé).

10.2.2 Définition du modèle de données

Nous commençons par définir la classe OwnerPIN. Puis nous nous intéressons aux différentes constantes utilisées dans l'application, avant de terminer par les attributs de classe.

La classe OwnerPIN

La classe OwnerPIN contient la valeur du code PIN, le nombre d'essais associés, ainsi que des valeurs booléennes permettant de savoir si le code PIN est validé, lors d'une identification, et défini, c'est-à-dire initialisé.

```
class OwnerPIN {

    //@ invariant pin >= 0 && pin <= 9999;
    short pin;

    final static byte PIN_MIN_TRY = 3;
    final static byte PIN_MAX_TRY = 15;

    //@ invariant triesLimit >= PIN_MIN_TRY && triesLimit <= PIN_MAX_TRY;
    byte triesLimit = 3;

    //@ invariant tries >= 0 && tries <= triesLimit;
    short tries;

    boolean validated, defined;

}
```

Les constantes de Demoney

Les constantes associées à la classe principale de Demoney sont les constantes liées aux niveaux d'accès pour les différents types de terminaux, et les constantes définissant les identifiants de l'applet (CLA) ou de chacune des instructions (INS).

Niveau d'accès Les constantes relatives aux différents niveaux d'accès sont données par le tableau 10.1.

Instructions Les constantes relatives aux instructions sont nommées par le nom de la méthode suffixée par `_INS` et dont les valeurs sont différentes pour chacune des méthodes. Par exemple, la constante correspondant à la méthode VERIFY_PIN est dénommée `VERIFY_PIN_INS`.

Type d'accès	Constante	Valeur
Public (défaut)	`PUBLIC_LVL`	0
Opérations de débit	`DEBIT_LVL`	1
Opérations de crédit	`CREDIT_LVL`	2
Administration	`ADMIN_LVL`	3

TAB. 10.1 – Constantes relatives aux niveaux d'accès

P1 et P2 Les valeurs des paramètres P1 et P2 sont utilisées dans toutes les méthodes. Les valeurs de ces paramètres sont fixées par des constantes ayant des valeurs prédéfinies dans le document de spécification. Le tableau 10.2 référence toutes les valeurs possibles pour P1 et P2 en fonction des méthodes auxquelles ils sont associés et indique leur signification.

On remarque un grand nombre de valeurs non-significatives (c'est-à-dire qu'elles représentent la seule valeur possible pour le paramètre) : P2 pour STORE_DATA, P2 pour INITIALIZE_TRANSACTION, P1 et P2 pour COMPLETE_TRANSACTION, P1 pour PIN_CHANGE_UNBLOCK, P2 pour EXTERNAL_AUTHENTICATE, P1 et P2 pour VERIFY_PIN, et P2 pour PUT_DATA). Les valeurs significatives sont soit des valeurs spécifiques données par le cahier des charges (P1 pour STORE_DATA), soit un ensemble de valeurs qui permettent d'identifier un comportement précis (P1 pour INITIALIZE_TRANSACTION, INITIALIZE_UPDATE, et PUT_DATA). Les valeurs de P1

Méthode	Paramètre	Constante	Valeur
STORE_DATA	P1	SD_P1_LAST_BLOCK	128
	P2	SD_P2	0
INITIALIZE_TRANSACTION	P1	IT_P1_DEBIT	0
		IT_P1_CREDIT_FROM_CASH	1
		IT_P1_CREDIT_FROM_BANK	2
	P2	IT_P2	0
COMPLETE_TRANSACTION	P1	CT_P1	0
	P2	CT_P2	0
PIN_CHANGE_UNBLOCK	P1	PCU_P1	0
	P2	PCU_P2_UNBLOCK	0
INITIALIZE_UPDATE	P1	DEBIT_KEY_NUM	1
		CREDIT_KEY_NUM	2
		ADMIN_KEY_NUM	3
	P2	IT_P2	0
EXTERNAL_AUTHENTICATE	P1	C_MAC	1
		C_MAC_R_MAC	16
	P2	EA_P2	0
VERIFY_PIN	P1	VERIFY_P1	0
	P2	VERIFY_P2	0
PUT_DATA	P1	PUT_MAX_BALANCE	1
		PUT_MAX_DEBIT	2
		PUT_PIN	3
	P2	PUT_DATA_P2	0

TAB. 10.2 – Valeurs des paramètres P1 et P2 en fonction des méthodes

pour EXTERNAL_AUTHENTICATE servent à désigner le niveau de sécurité de la transaction (C_MAC ou C_MAC_R_MAC).

Intéressons-nous désormais aux variables d'états, à savoir les attributs de la classe Demoney.

Les attributs de classe

Nous présentons ici les attributs de classe et les invariants qui s'appliquent sur ceux-ci.

Le code PIN permet de savoir, à travers l'utilisation d'un objet OwnerPIN les informations relatives à l'identification du porteur du porte-monnaie.

```
//@ invariant pin != null;
private OwnerPIN pin;
```

Une variable booléenne est utilisée pour savoir si le porte-monnaie a été déjà été personnalisé ou non.

```
private boolean personnalized;
```

Les valeurs symbolisant les plafonds du solde et du débit maximal autorisés sont conservés dans deux variables de type entier court. Ces deux valeurs sont toujours strictement positives.

```
//@ invariant maxBalance > 0;
private short maxBalance;

//@ invariant maxDebit > 0;
private short maxDebit;
```

La valeur du solde du porte-monnaie est également conservée dans un attribut de type entier court. Le solde est toujours positif et inférieur au plafond autorisé.

```
//@ invariant balance >= 0 && balance <= maxBalance;
private short balance;
```

Le niveau d'accès courant de la carte est également stocké.

```
//@ invariant accessLevel >= PUBLIC_LVL &&
              accessLevel <= ADMIN_LVL ;              */
private byte accessLevel;
```

On ajoute deux attributs conservant les séquences de transaction et d'authentification, qui permettent de savoir si une séquence d'un de ces genres a été initiée.

```
private short transactionSequence;
private byte authenticateSequence;
```

Si une séquence de transaction est initiée, la valeur de `transactionSequence` est différentes de 0 ; plus précisément, cet attribut a pour valeur le montant de la transaction en cours. De la même manière, si une séquence d'authentification est initiée, `authenticateSequence` a pour valeur le niveau d'accès demandé lors de l'authentification.

```
/*@ invariant authenticateSequence >= PUBLIC_LVL &&
              authenticateSequence <= ADMIN_LVL ;     */
```

Pour finir, deux séquences ne peuvent être initiées en même temps.

```
//@ invariant !(authenticateSequence != 0 && transactionSequence != 0);
```

Nous décrivons pour finir le constructeur de classe qui donne les valeurs initiales des attributs de la classe Demoney.

Constructeur de classe

Le constructeur de classe initialise les valeurs des attributs de classe à leur valeur par défaut (dans le cadre de l'application).

```
/*@ normal_behavior
  @    assignable this.*;
  @    ensures \fresh(pin) && pin.pin == 0 && pin.tries == 0 &&
  @            pin.validated == false && pin.defined == false;
  @    ensures personnalized == false &&
  @            maxBalance == 1 && maxDebit == 1 &&
  @            balance == 0 &&
  @            accessLevel == PUBLIC_LVL &&
  @            authenticateSequence == PUBLIC_LVL &&
  @            transactionSequence == 0;
  @*/
public Demoney() { ... }
```

Nous décrivons à présent les méthodes et nous donnons leur fonctionnement.

10.2.3 Définition des méthodes

Nous donnons la modélisation des méthodes dans l'ordre où celles-ci peuvent être utilisées dans le processus de vie de la carte, i.e., les étapes de personnalisation, le processus d'authentification, le processus de transaction, et la vérification du code PIN. Pour simplifier la modélisation, les paramètres CLA et INS spécifiant respectivement l'identifiant de l'applet et l'identifiant de l'instruction exécutée, sont volontairement omis, car ils n'apportent rien à la modélisation. En effet, ces derniers identifient la méthode appelée et seront retranscrits dans la phase de réification sans ambiguïté. Les comportements exceptionnels sont déclenchés lorsque les conditions d'appel de la méthode sont incorrectes. Ainsi, chaque comportement exceptionnel est préconditionné par la négation des préconditions normales.

Les différents cas d'erreurs sont décrits par deux exceptions qui spécifient soit que l'appel de la méthode est incorrect vis-à-vis de l'état actuel de la carte (IllegalUseException), soit que les valeurs des paramètres sont incorrects (WrongParameterException).

Ces différents cas peuvent être vus comme des cas de tests de robustesse. En effet, les méthodes sont supposées être appelées de n'importe quel état du système, elles présentent donc systématiquement deux facettes : le comportement normal qui correspond à un appel de méthode correct vis-à-vis des pré-requis du cahier des charges, et des comportements exceptionnels qui indiquent en quoi l'appel à la méthode est illicite.

La personnalisation

La première méthode permettant la personnalisation est la méthode PUT_DATA. Cette méthode ne peut être activée que si la personnalisation de la carte n'est pas terminée, et si l'utilisateur est administrateur. Le paramètre P1 une valeur permettant de savoir quel attribut est utilisé, entre le plafond du solde, le plafond associé au débit et le code PIN. Dans ce dernier cas, le paramètre P2 spécifie le nombre d'essais associé au code PIN.

```
/*@ normal_behavior
  @    requires personnalized == false &&
  @             accessLevel == ADMIN_LVL;
  @    {|
  @        requires P1 == PUT_MAX_BALANCE && P2 == PUT_DATA_P2 && data > 0;
  @        assignable maxBalance;
  @        ensures maxBalance == data;
  @    also
  @        requires P1 == PUT_MAX_DEBIT && P2 == PUT_DATA_P2 && data > 0;
  @        assignable maxDebit;
  @        ensures maxDebit == data;
  @    also
  @        requires P1 == PUT_PIN &&
  @                 P2 >= OwnerPIN.PIN_MIN_TRY && P2 <= OwnerPIN.PIN_MAX_TRY &&
  @                 data >= 0 && data <= 9999;
  @        assignable pin.pin, pin.tries;
  @        ensures pin.pin == data && pin.tries == P2;
  @    |}
  @ also
  @   exceptional_behavior
  @      requires personnalized == true || accessLevel != ADMIN_LVL;
  @      assignable \nothing;
  @      signals (IllegalUseException) true;
  @   also
  @      requires personnalized == false && accessLevel == ADMIN_LVL &&
  @               (P1 != PUT_MAX_BALANCE && P1 != PUT_MAX_DEBIT && P1 != PUT_PIN)
  @      assignable \nothing;
  @      signals (WrongParameterException) true;
  @   also
  @      requires personnalized == false && accessLevel == ADMIN_LVL &&
  @               P1 == PUT_MAX_BALANCE && (P2 != PUT_DATA_P2 || data <= 0)
  @      assignable \nothing;
  @      signals (WrongParameterException) true;
  @   also
  @      requires personnalized == false && accessLevel == ADMIN_LVL &&
  @               P1 == PUT_MAX_DEBIT && (P2 != PUT_DATA_P2 || data <= 0)
  @      assignable \nothing;
  @      signals (WrongParameterException) true;
  @   also
  @      requires personnalized == false && accessLevel == ADMIN_LVL &&
  @               P1 == PUT_PIN && ((P2 < OwnerPIN.PIN_MIN_TRY || P2 > OwnerPIN.PIN_MAX_TRY) ||
  @                                 (data < 0 || data > 9999));
  @      assignable \nothing;
  @      signals (WrongParameterException) true;
  @*/
public PUT_DATA(byte P1, byte P2, short data)
                     throws IllegalUseException,WrongParameterException { ... }
```

La personnalisation est fixée par la commande STORE_DATA. Celle-ci ne peut être activée que par un administrateur et uniquement si aucune séquence d'authentification ou de transaction n'est en cours, et si le code PIN de la carte a été défini. Si cette méthode n'est pas appelée dans les conditions requises, elle restaure un état neutre du système où aucune séquence d'authentification ou de transaction n'a été initiée.

```
/*@ normal_behavior
  @    requires     personnalized == false && accessLevel == ADMIN_LVL &&
  @                 authenticateSequence == PUBLIC_LVL && transactionSequence == 0 &&
  @                 P1 == SD_P1_LAST_BLOCK && P2 == SD_P2 && pin.defined == true;
  @    assignable   personnalized;
  @    ensures      personnalized == true;
  @ also
  @    exceptional_behavior
  @       requires   personnalized != false || accessLevel != ADMIN_LVL ||
  @                  authenticateSequence != PUBLIC_LVL || transactionSequence != 0 || pin.defined == false;
  @       assignable transactionSequence, authenticateSequence;
  @       signals (IllegalUseException)
  @                  authenticateSequence == PUBLIC_LVL && transactionSequence == 0;
  @     also
  @       requires   personnalized == false && accessLevel == ADMIN_LVL &&
  @                  authenticateSequence == PUBLIC_LVL && transactionSequence == 0 && pin.defined == true;
  @       requires    P1 != SD_P1_LAST_BLOCK || P2 != SD_P2;
  @       assignable  transactionSequence, authenticateSequence;
  @       signals (WrongParameterException)
  @                   authenticateSequence == PUBLIC_LVL && transactionSequence == 0;
  @*/
public void STORE_DATA(byte P1, byte P2)
                   throws IllegalUseException,WrongParameterException { ... }
```

Nous nous intéressons désormais à l'authentification de l'utilisateur de la carte.

Le mécanisme d'authentification

L'authentification s'effectue en deux étapes. Tout d'abord, il faut choisir le niveau d'accès requis à travers l'appel à la méthode INITIALIZE_UPDATE, puis valider cette authentification avec un appel à EXTERNAL_AUTHENTICATE.

La méthode INITIALIZE_UPDATE permet d'initier l'authentification. Pour passer en mode *credit* ou *debit*, il est requis que la carte soit au préalable personnalisée. Le passage en mode administrateur peut se faire quelque soit le degré de personnalisation de la carte. On distingue quatre cas, correspondant aux quatre types d'accès. Le passage au niveau *public* correspond au mode par défaut, i.e., l'exclusion des 3 cas précédents.

```
/*@ normal_behavior
  @    requires     personnalized == true &&
  @                 P1 == DEBIT_KEY_NUM && P2 == KEY_INDEX;
  @    assignable   authenticateSequence;
  @    ensures      authenticateSequence == DEBIT_LVL;
  @
  @ also
  @    requires     personnalized == true &&
  @                 P1 == CREDIT_KEY_NUM && P2 == KEY_INDEX
  @    assignable   authenticateSequence;
  @    ensures      authenticateSequence == CREDIT_LVL;
  @ also
  @    requires     P1 == ADMIN_KEY_NUM && P2 == KEY_INDEX
  @    assignable   authenticateSequence;
  @    ensures      authenticateSequence == ADMIN_LVL;
  @ also
```

```
  @    requires     P2 != KEY_INDEX || ((personnalized == false &&
  @                    (P1 != CREDIT_KEY_NUM || P1 != DEBIT_KEY_NUM)) && P1 != ADMIN_KEY_NUM);
  @    assignable   authenticateSequence;
  @    ensures      authenticateSequence == PUBLIC_LVL;
  @*/
public void INITIALIZE_UPDATE(byte P1, byte P2) { ... }
```

La méthode EXTERNAL_AUTHENTICATE suit l'exécution de INITIALIZE_UPDATE, et sert à valider le changement d'accès précédemment initié. L'appel peut être réalisé avec deux valeurs pour le paramètre `p1` désignant le niveau de sécurité de la transaction.

```
/*@ normal_behavior
  @    requires     (P1 == C_MAC || P1 == C_MAC_R_MAC) &&
  @                 P2 == EA_P2 &&
  @                 authenticateSequence != PUBLIC_LVL;
  @    assignable   accessLevel, authenticateSequence, transactionSequence;
  @    ensures      accessLevel == \old(authenticateSequence) &&
  @                 authenticateSequence == PUBLIC_LVL &&
  @                 transactionSequence == 0;
  @ also
  @   exceptional_behavior
  @      requires   (P1 == C_MAC || P1 == C_MAC_R_MAC) &&
  @                 P2 == EA_P2;
  @      requires   authenticateSequence == PUBLIC_LVL;
  @      assignable authenticateSequence;
  @      signals (IllegalUseException)
  @                 authenticateSequence == PUBLIC_LVL;
  @   also
  @      requires   (P1 != C_MAC && P1 != C_MAC_R_MAC) ||
  @                 P2 != EA_P2;
  @      requires   authenticateSequence != PUBLIC_LVL;
  @      assignable authenticateSequence;
  @      signals (WrongParameterException)
  @                 authenticateSequence == PUBLIC_LVL;
  @*/
public void EXTERNAL_AUTHENTICATE(byte P1, byte P2)
                throws IllegalUseException,WrongParameterException { ... }
```

La séquence de personnalisation qui échoue replace le système dans un état neutre. Si celle-ci réussit, le niveau d'accès est passé au niveau précédemment demandé, temporairement conservé dans l'attribut `authenticateSequence`.

Le mécanisme de transaction

Comme pour l'authentification, le mécanisme de transaction se décompose en deux commandes qui se succèdent : INITIALIZE_TRANSACTION et COMPLETE_TRANSACTION.

La méthode INITIALIZE_TRANSACTION n'est activable que si la carte est déjà personnalisée, et qu'aucune séquence de transaction ou d'authentification n'est déjà initiée. Cette méthode vérifie que le niveau d'accès courant autorise l'action à effectuer. Dans le cas d'un débit ou d'un crédit, la vérification des dépassements des plafonds respectifs est également effectuée. De plus, un crédit à partir d'un compte bancaire requiert que le code PIN ait été au préalable validé.

En cas d'appel illicite, la méthode restaure un état neutre du système dans lequel aucune séquence d'authentification ou de transaction n'est en cours.

```
/*@ normal_behavior
  @    requires  personnalized == true && P2 == IT_P2 &&
  @              authenticateSequence == PUBLIC_LVL && transactionSequence == 0;
  @    {|
  @       requires    P1 == IT_P1_DEBIT && accessLevel == DEBIT_LVL && data > 0 &&
  @                   data <= maxDebit && data <= balance;
  @       assignable  transactionSequence;
  @       ensures     transactionSequence == (short) - data;
  @
  @    also
  @       requires    P1 == IT_P1_CREDIT_FROM_CASH && accessLevel == CREDIT_LVL && data > 0 &&
  @                   (balance + data) <= maxBalance;
  @       assignable  transactionSequence;
  @       ensures     transactionSequence == data;
  @
  @    also
  @       requires    P1 == IT_P1_CREDIT_FROM_BANK &&
  @                   accessLevel == CREDIT_LVL && data > 0 &&
  @                   pin.validated == true &&
  @                   (balance + data) <= maxBalance;
  @       assignable  transactionSequence, pin.validated;
  @       ensures     transactionSequence == data && pin.validated == false;
  @    |}
  @ also
  @    exceptional_behavior
  @       requires    personnalized == false || authenticateSequence != PUBLIC_LVL;
  @       assignable  authenticateSequence, transactionSequence;
  @       signals (IllegalUseException)
  @                   authenticateSequence == PUBLIC_LVL && transactionSequence == 0;
  @    also
  @       requires    (P1 != IT_P1_CREDIT_FROM_BANK && P1 != IT_P1_CREDIT_FROM_CASH &&
  @                    P1 != IT_P1_DEBIT) || P2 != IT_P2 || data <= 0;
  @       assignable  authenticateSequence, transactionSequence;
  @       signals (WrongParameterException)
  @                   authenticateSequence == PUBLIC_LVL && transactionSequence == 0;
  @     also
  @        requires   (P1 == IT_P1_DEBIT && accessLevel != DEBIT_LVL) ||
  @                   (P1 == IT_P1_CREDIT_FROM_CASH && (balance + data > maxBalance ||
  @                                                     accessLevel != CREDIT_LVL) ||
  @                   (P1 == IT_P1_CREDIT_FROM_BANK && (pin.validated != true ||
  @                                                     balance + data > maxBalance ||
  @                                                     accessLevel != CREDIT_LVL);
  @       assignable  authenticateSequence,transactionSequence;
  @       signals (WrongParameterException)
  @                   authenticateSequence == PUBLIC_LVL && transactionSequence == 0;
  @*/
public void INITIALIZE_TRANSACTION(byte P1, byte P2, short data)
               throws IllegalUseException, WrongParameterException { ... }
```

La transaction initiée est validée lorsque la méthode COMPLETE_TRANSACTION est invoquée. Celle-ci ajoute au solde du porte-monnaie le montant temporairement conservé dans l'attribut `transactionSequence`.

```
/*@ normal_behavior
  @    requires    personnalized == true &&
  @                P1 == CT_P1 && P2 == CT_P2 &&
  @                transactionSequence != 0;
  @    assignable  balance, transactionSequence;
  @    ensures     balance == \old(balance) + \old(transactionSequence) &&
  @                transactionSequence == 0;
  @ also
  @   exceptional_behavior
  @      requires   personnalized == false || transactionSequence == 0;
  @      requires   P1 == CT_P1 && P2 == CT_P2;
  @      assignable authenticateSequence;
  @      signals (IllegalUseException) authenticateSequence == PUBLIC_LVL ;
```

```
  @   also
  @      requires   personnalized == true && transactionSequence != 0;
  @      requires   P1 != CT_P1 || P2 != CT_P2;
  @      assignable authenticateSequence;
  @      signals (WrongParameterException) authenticateSequence == PUBLIC_LVL ;
  @*/
public void COMPLETE_TRANSACTION(byte P1, byte P2) throws Exception { ... }
```

Comme nous avons pu le constater, le crédit à partir de la banque nécessite l'identification par code PIN au préalable. Décrivons, pour en terminer avec la modélisation, les principes de fonctionnement du code PIN.

La vérification du code PIN

La vérification du code PIN s'effectue par l'intermédiaire de la méthode VERIFY_PIN. Si le code PIN est amené à être bloqué, suite à plusieurs essais infructueux successifs, celui-ci peut-être débloqué par l'intermédiaire de la méthode PIN_CHANGE_UNBLOCK. Cette méthode est également utilisée pour changer le code PIN, et le nombre de tentatives qui lui est associé. Ces deux opérations sont réalisées par un administrateur.

La vérification du code PIN consiste à fournir un code PIN et à vérifier que ce code correspond au code associé à la carte. Si ce n'est pas le cas, le nombre d'essais est décrémenté, jusqu'au moment où celui-ci vaut 0. Il passe alors en statut bloqué.

```
/*@ normal_behavior
  @    requires personnalized == true && accessLevel == CREDIT_LVL &&
  @             P1 == VERIFY_P1 && P2 == VERIFY_P2 && pin.defined == true &&
  @             pin.tries != 0 && data >= 0 && data <= 9999;
  @    {|
  @       requires    data == pin.pin;
  @       assignable  pin.validated, pin.tries;
  @       ensures     pin.validated == true && pin.tries == pin.triesLimit;
  @    also
  @       requires    data != pin.pin;
  @       assignable  pin.tries;
  @       ensures     pin.tries == \old(pin.tries) - 1;
  @    |}
  @ also
  @    exceptional_behavior
  @       requires    P1 == VERIFY_P1 && P2 == VERIFY_P2;
  @       requires    personnalized == false || accessLevel != CREDIT_LVL ||
  @                   pin.defined == false || pin.tries == 0;
  @       assignable  authenticateSequence;
  @       signals (IllegalUseException)
  @                   authenticateSequence == PUBLIC_LVL;
  @    also
  @       requires    P1 != VERIFY_P1 || P2 != VERIFY_P2;
  @       requires    personnalized == true && accessLevel == CREDIT_LVL &&
  @                   pin.defined != false && pin.tries != 0;
  @       assignable  authenticateSequence;
  @       signals (WrongParameterException)
  @                   authenticateSequence == PUBLIC_LVL;
  @*/
public void VERIFY_PIN(byte P1, byte P2, short data)
                throws IllegalUseException,WrongParameterException { ... }
```

Si le code PIN est bloqué, ou pour le changer, on applique la méthode PIN_CHANGE_UNBLOCK. L'appel à cette méthode avec le paramètre P2 valué à PCU_P2_UNBLOCK indique que la méthode sert à débloquer le code PIN. Sinon, P2 doit représenter le nouveau nombre d'essais associé au nouveau code PIN, donné par le paramètre "data".

```
/*@ normal_behavior
  @    requires    personnalized == true && P1 == PCU_P1 && accessLevel == ADMIN_LVL;
  @    {|
  @       requires    P2 == PCU_P2_UNBLOCK && pin.tries == 0;
  @       assignable  pin.tries, transactionSequence, authenticateSequence;
  @       ensures     pin.tries == pin.triesLimit &&
  @                   transactionSequence == 0 && authenticateSequence == PUBLIC_LVL;
  @    also
  @       requires    (P2 >= pin.MIN_PIN_TRY && P2 <= pin.MAX_PIN_TRY) &&
  @                   data >= 0 && data <= 9999 && pin.tries > 0;
  @       assignable  pin.pin, pin.tries, pin.triesLimit, transactionSequence, authenticateSequence;
  @       ensures     pin.pin == data && transactionSequence == 0 &&
  @                   authenticateSequence == PUBLIC_LVL && pin.tries == P2 &&
  @                   pin.triesLimit == P2;
  @    |}
  @ also
  @    exceptional_behavior
  @      requires    personnalized == false || accessLevel != ADMIN_LVL;
  @      requires    (P2 == PCU_P2_UNBLOCK && pin.tries == 0) ||
  @                  (P2 >= MIN_PIN_TRY_LIMIT && P2 <= MAX_PIN_TRY_LIMIT &&
  @                   data >= 0 && data <= 9999);
  @      assignable  authenticateSequence, transactionSequence;
  @      signals (IllegalUseException)
  @                  authenticateSequence == PUBLIC_LVL &&
  @                  transactionSequence == 0 ;
  @    also
  @      requires    personnalized == true && accessLevel == ADMIN_LVL;
  @      requires    (P2 != PCU_P2_UNBLOCK || pin.tries >= 0) &&
  @                  (P2 < MIN_PIN_TRY_LIMIT || P2 > MAX_PIN_TRY_LIMIT ||
  @                   data < 0 || data > 9999);
  @      assignable  authenticateSequence, transactionSequence;
  @      signals (WrongParameterException)
  @                  authenticateSequence == PUBLIC_LVL &&
  @                  transactionSequence == 0 ;

  @*/
public void PIN_CHANGE_UNBLOCK(byte P1, byte P2, short data)
                  throws IllegalUseException,WrongParameterException { ... }
```

Validation du modèle à l'animateur JML-Testing-Tools

Ce modèle sert de référence pour la génération de tests. Sa mise au point a été réalisée avec l'appui de l'animateur qui a permis de mettre en évidence divers oublis et erreurs au fur et à mesure de la conception.

Les erreurs les plus courantes qui ont été réalisées lors de la mise au point du modèle, et corrigées par la suite, sont :

- oubli de mise à jour d'un attribut – par exemple, ne pas réinitialiser `pin.tries` suite à la vérification du code PIN (méthode VERIFY_PIN).
- oubli de comportements – par exemple, oublier un des cas d'erreurs de PUT_DATA ;
- écriture d'une précondition trop faible par rapport aux cahier des charges – par exemple, permettre de débloquer un code PIN alors que le nombre d'essais n'est pas égal à 0 ;
- échange de deux attributs dans un prédicat – par exemple, (`authenticateSequence` et `transactionSequence`) ;

Nous nous intéressons à présent au processus de génération de tests à partir de cette modélisation.

10.3 Génération de tests pour l'application

Cette partie reprend le modèle précédemment écrit pour générer des tests. Nous décrivons dans un premier temps les buts aux limites calculés par notre approche puis nous présentons une partie des tests générés. Pour finir, nous illustrons le genre d'erreurs détectés pour une implantation erronée de l'exemple.

Pour simplifier la lecture, nous ne nous intéressons qu'à certaines méthodes, les plus représentatives, en donnant les cibles de tests, les cas de test associés seront donnés en annexe B.

10.3.1 Extraction des objectifs de tests

Nous présentons ici l'extraction des objectifs de tests. Les objectifs de tests dépendent des comportements et des réécritures choisies pour les disjonctions. Ils dépendent également des buts aux limites calculés.

Reprenons la méthode PUT_DATA décrite en section précédente. Les comportements entraînant une terminaison normale sont tous exclusifs et ne contiennent pas de disjonction. Le tableau 10.3 donne les objectifs de tests et les prédicats de spécialisation associés au comportement normaux, pour chacune des réécritures employées.

Cpt	Partie avant	P_{spe}
cpt_1	`personnalized == false && accessLevel == ADMIN_LVL &&` `P1 == PUT_MAX_BALANCE && P2 == PUT_DATA_P2 && data > 0`	`data == 1`
		`data == 32767`
cpt_2	`personnalized == false && accessLevel == ADMIN_LVL &&` `P1 == PUT_MAX_DEBIT && P2 == PUT_DATA_P2 && data > 0`	`data == 1`
		`data == 32767`
cpt_3	`personnalized == false && accessLevel == ADMIN_LVL &&` `P1 == PUT_PIN && data >= 0 && data <= 9999 &&` `P2 >= OwnerPIN.PIN_MIN_TRY && P2 <= OwnerPIN.PIN_MAX_TRY`	`P2 == 3 &&` `data == 0`
		`P2 == 15 &&` `data == 9999`

TAB. 10.3 – Cibles de tests pour les comportements normaux de PUT_DATA

Les comportements exceptionnels comportent des disjonctions dont les différentes réécritures produisent différents objectifs de tests. Le tableau 10.4 donne les objectifs de tests et les prédicats de spécialisation associés au premier comportement exceptionnel.

Les cas d'erreurs représentés par les comportements exceptionnels des différentes méthodes sont sujets à proposer un grand nombre de cibles de tests, en fonction des différentes réécritures. Nous présentons dans le tableau 10.5 le nombre de buts aux limites visant à activer des comportements exceptionnels pour chacune des méthodes en fonction des différentes réécritures. Chaque résultat, présenté sous la forme A/B, donne le nombre de cibles consistantes A par rapport au nombre de cibles générées B. Il est à noter que les valeurs de A et B prennent en compte les réécritures et la minimisation/maximisation des attributs ou des paramètres.

Nous donnons, pour finir, l'ensemble des cibles de tests extraites des différents comportements normaux des méthodes INITIALIZE_TRANSACTION (tableau 10.6) et VERIFY_PIN (tableau 10.7). Les disjonctions sont traitées par une réécriture 3 qui confère une couverture exclusive de chacun des différents cas satisfaisant la disjonction.

Cpt	Réécriture	Cible	P_{spe}
cpt_7	1	`personnalized == true || accessLevel != ADMIN_LVL`	`accessLevel == 0`
			`accessLevel == 2`
	2	`personnalized == true`	
		`accessLevel != ADMIN_LVL`	`accessLevel == 0`
			`accessLevel == 2`
	3	`personnalized == true && accessLevel == ADMIN_LVL`	
		`personnalized != true && accessLevel != ADMIN_LVL`	`accessLevel == 0`
			`accessLevel == 2`
	4	`personnalized == true && accessLevel == ADMIN_LVL`	
		`personnalized != true && accessLevel != ADMIN_LVL`	`accessLevel == 0`
			`accessLevel == 2`
		`personnalized == true && accessLevel != ADMIN_LVL`	`accessLevel == 0`
			`accessLevel == 2`

TAB. 10.4 – Exemple de cibles de tests pour un comportement exceptionnel de PUT_DATA

Nous nous intéressons à présent au calcul de séquences de tests visant à atteindre les buts aux limites ainsi calculés.

10.3.2 Calcul des séquences de tests

Le calcul de séquences de tests se réalise initialement à deux niveaux : les séquences de tests créées à l'aide de l'animateur JML-TESTING-TOOLS, et les séquences calculées automatiquement. Comme nous allons le voir, le calcul automatique peut échouer à parvenir à atteindre une cible de test à une profondeur suffisante et dans un temps raisonnable. Pour résoudre ce problème, nous proposons un couplage de l'animateur et du générateur de tests, de manière à calculer semi-automatiquement le cas de test.

Création de séquences de tests avec l'animateur

L'étude de cas tirée du monde de la JavaCard, offre la possibilité de réaliser de nombreux enchaînements de méthodes, pour exercer le mécanisme de transaction, et/ou d'authentification de la carte.

Méthode	Réécriture 1	Réécriture 2	Réécriture 3	Réécriture 4
PUT_DATA	10/10	20/20	20/20	26/28
STORE_DATA	4/4	14/14	14/14	16/16
EXTERNAL_AUTHENTICATE	4/4	6/6	6/6	8/9
INITIALIZE_TRANSACTION	6/6	15/15	15/15	17/18
COMPLETE_TRANSACTION	3/3	6/6	7/7	10/10
VERIFY_PIN	4/4	9/9	9/9	13/13
PIN_CHANGE_UNBLOCK	4/4	25/25	25/25	25/29

TAB. 10.5 – Nombre de cibles de tests pour les comportements exceptionnels en fonction des réécritures

Cpt	Partie avant	P_{spe}
cpt_1	`personnalized == true && authenticateSequence == PUBLIC_LVL && transactionSequence == 0 && p2 == IT_P2 && P1 == IT_P1_DEBIT && accessLevel == DEBIT_LVL && data > 0 && data <= maxDebitAmount && data <= balance`	`data == 1 && maxDebitAmount == 1 && balance == 1`
		`data == 32767 && maxDebitAmount == 32767 && balance == 32767`
cpt_2	`personnalized == true && authenticateSequence == PUBLIC_LVL && transactionSequence == 0 && p2 == IT_P2 && P1 == IT_P1_CREDIT_FROM_CASH && accessLevel == CREDIT_LVL && data > 0 && (balance + data) <= maxBalance`	`data == 1 && maxBalance == 1 && balance == 0`
		`data == 32767 && maxBalance == 32767 && balance == 0`
cpt_3	`personnalized == true && authenticateSequence == PUBLIC_LVL && transactionSequence == 0 && p2 == IT_P2 && P1 == IT_P1_CREDIT_FROM_BANK && accessLevel == CREDIT_LVL && data > 0 && (balance + data) <= maxBalance && pin.validated == true`	`data == 1 && maxBalance == 1 && balance == 0`
		`data == 32767 && maxBalance == 32767 && balance == 0`

TAB. 10.6 – Cibles de tests pour les comportements normaux de INITIALIZE_TRANSACTION

Cpt	Partie avant	P_{spe}
cpt_1	`personnalized == true && accessLevel == CREDIT_LVL P1 == VERIFY_P1 && P2 == VERIFY_P2 && pin.defined == true && pin.tries != 0 && data >= 0 && data <= 9999 data == pin.pin`	`data == 0 && pin.tries == 1 && pin.pin == 0`
		`data == 9999 && pin.tries == 15 && pin.pin == 9999`
cpt_2	`personnalized == true && accessLevel == CREDIT_LVL P1 == VERIFY_P1 && P2 == VERIFY_P2 && pin.defined == true && pin.tries != 0 && data >= 0 && data <= 9999 data != pin.pin`	`data == 0 && pin.tries == 1 && pin.pin == 1`
		`data == 9999 && pin.tries == 15 && pin.pin == 9998`

TAB. 10.7 – Cibles de tests pour les comportements normaux de VERIFY_PIN

Dans un premier temps, nous utilisons le modèle validé pour construire des séquences de tests à l'animateur. Si la construction de ces séquences nécessite l'expertise de l'ingénieur validation pour choisir les enchaînements de méthodes, le calcul des données de tests peut être réalisé de manière automatique et systématique par notre approche.

Nous proposons, parmi les milliers possibles, les deux scénarios donnés en figure 10.1. Ces deux scénarios commencent avec la phase de personnalisation de la carte qui fixe le solde maximal à 10000, le débit maximal à 5000 et le code PIN à la valeur 1234 avec 3 essais pour la vérification de celui-ci. Le premier scénario effectue un crédit puis un débit, en ayant au préalable changé le niveau d'accès pour satisfaire l'autorisation requise pour chacune de ces transactions. Le second s'intéresse à bloquer la carte suite à 3 essais infructueux et à la débloquer, puis à changer le code PIN, en fixant à cette occasion le nombre d'essais de déblocage à 4. Au passage, ce cas de test cherche à s'assurer qu'après 3 essais erronés, il est impossible d'exécuter VERIFY_PIN, même avec un code correct, sans déclencher d'exception.

```
Demoney d = new Demoney();
d.INITIALIZE_UPDATE((byte) 3,(byte) 1);
d.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);
d.PUT_DATA((byte) 1,(byte) 0,(short) 10000);
d.PUT_DATA((byte) 2,(byte) 0,(short) 5000);
d.PUT_DATA((byte) 3,(byte) 3,(short) 1234);
d.STORE_DATA((byte) 80,(byte) 0);
d.INITIALIZE_UPDATE((byte) 2,(byte) 1);
d.EXTERNAL_AUTHENTICATE((byte) 11,(byte) 0);
d.VERIFY_PIN((byte) 0,(byte) 0,(short) 4321);
d.VERIFY_PIN((byte) 0,(byte) 0,(short) 1234);
d.INITIALIZE_TRANSACTION((byte) 2,(byte) 0,
                         (short) 6000);
d.COMPLETE_TRANSACTION((byte) 0,(byte) 0);
d.INITIALIZE_UPDATE((byte) 1,(byte) 1);
d.EXTERNAL_AUTHENTICATE((byte) 11,(byte) 0);
d.INITIALIZE_TRANSACTION((byte) 0,(byte) 0,
                        (short) 4000);
d.COMPLETE_TRANSACTION((byte) 0,(byte) 0);
```

```
Demoney d = new Demoney();
d.INITIALIZE_UPDATE((byte) 3,(byte) 1);
d.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);
d.PUT_DATA((byte) 1,(byte) 0,(short) 10000);
d.PUT_DATA((byte) 2,(byte) 0,(short) 5000);
d.PUT_DATA((byte) 3,(byte) 3,(short) 1234);
d.STORE_DATA((byte) 80,(byte) 0);
d.INITIALIZE_UPDATE((byte) 2,(byte) 1);
d.EXTERNAL_AUTHENTICATE((byte) 11,(byte) 0);
d.VERIFY_PIN((byte) 0,(byte) 0,(short) 2143);
d.VERIFY_PIN((byte) 0,(byte) 0,(short) 3421);
d.VERIFY_PIN((byte) 0,(byte) 0,(short) 4213);
try {
   d.VERIFY_PIN((byte) 0, (byte) 0, (short) 1234);
   throw new JMLTTUnraisedException(
               ''IllegalUseException'');
}
catch (IllegalUseException _exc) { }
d.INITIALIZE_UPDATE((byte) 3,(byte) 1);
d.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);
d.PIN_CHANGE_UNBLOCK((byte) 0,(byte) 0,(short) 0);
d.PIN_CHANGE_UNBLOCK((byte) 0,(byte) 4,(short) 4321);
```

FIG. 10.1 – Deux scénarios conçus avec l'animateur JML-TESTING-TOOLS

Outre la possibilité d'animer et ainsi de valider le modèle, l'animateur permet de créer de manière semi-automatique des cas de tests Java. Nous nous intéressons à présent à la génération automatique des séquences de tests visant à atteindre les buts aux limites précédemment identifiés.

Calcul automatique

Les séquences de tests calculées automatiquement visent à atteindre un état satisfaisant l'objectif de test fixé. La limitation dans la profondeur instaurée pour le calcul de préambule restreint les cibles de tests concrètement atteignables. De plus, la validation des authentifications et des transactions passe par un état temporaire, qui représente

un "minimum local" ralentissant la convergence de l'algorithme de calcul des tests. Pour une profondeur de recherche de 6, seuls les comportements des méthodes PUT_DATA, STORE_DATA, INITIALIZE_UPDATE et EXTERNAL_AUTHENTICATE sont atteignables en un temps très satisfaisant : en réécriture 2, en ne considérant que les comportements normaux, les 6 cibles de test de PUT_DATA sont couverts en un total de 18 s, la cible de test de STORE_DATA est couverte en 49 s, les 13 cibles de test issues de la méthode INITIALIZE_UPDATE sont toutes couvertes en 50 s, et les 4 cibles extraites de la méthode EXTERNAL_TRANSACTION sont couvertes en 74 s. Ces mesures ont été effectuées sur un Pentium III cadencé à 600 MHz avec 256 Mo de RAM.

Nous présentons dans le tableau 10.8, les cas de tests visant à atteindre les cibles relatives à la méthode VERIFY_PIN, données le tableau 10.7. Dans ce tableau, les comportements se décomposent en deux : la partie minimisée (dénotée par cpt^{max}) et la partie maximisée (dénotée par cpt^{max}). Comme nous pouvons le constater, la taille des séquences de test produites est relativement élevée (entre 8 et 10 opérations), ce qui se ressent sur le temps de calcul (plusieurs heures sont nécessaires pour couvrir les 4 cibles). Ce phénomène est accentué par la présence de minima locaux dans le graphe d'exécution du modèle, qui ralentit la recherche. Les mécanismes d'authentification et de transaction, réalisés en deux

Cpt	Séquence d'exécution	Cas de test

cpt_1^{min}	`Demoney var0 = new Demoney();`
	`var0.INITIALIZE_UPDATE((byte) 3,(byte) 1);`
	`var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);`
	`var0.PUT_DATA((byte) 3,(byte) 3,(short) 0);`
	`var0.STORE_DATA((byte) 80,(byte) 0);`
	`var0.INITIALIZE_UPDATE((byte) 2,(byte) 1);`
	`var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);`
	`var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 1);`
	`var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 1);`
	`var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 0);`
		(1)

cpt_1^{max}	`Demoney var0 = new Demoney();`
	`var0.INITIALIZE_UPDATE((byte) 3,(byte) 1);`
	`var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);`
	`var0.PUT_DATA((byte) 3,(byte) 15,(short) 9999);`
	`var0.STORE_DATA((byte) 80,(byte) 0);`
	`var0.INITIALIZE_UPDATE((byte) 2,(byte) 1);`
	`var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);`
	`var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 9999);`
		(2)

cpt_2^{min}	`Demoney var0 = new Demoney();`
	`var0.INITIALIZE_UPDATE((byte) 3,(byte) 1);`
	`var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);`
	`var0.PUT_DATA((byte) 3,(byte) 3,(short) 0);`
	`var0.STORE_DATA((byte) 80,(byte) 0);`
	`var0.INITIALIZE_UPDATE((byte) 2,(byte) 1);`
	`var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);`
	`var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 1);`
	`var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 1);`
	`var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 1);`
		(3)

cpt_2^{max}	`Demoney var0 = new Demoney();`
	`var0.INITIALIZE_UPDATE((byte) 3,(byte) 1);`
	`var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);`
	`var0.PUT_DATA((byte) 3,(byte) 15,(short) 9999);`
	`var0.STORE_DATA((byte) 80,(byte) 0);`
	`var0.INITIALIZE_UPDATE((byte) 2,(byte) 1);`
	`var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);`
	`var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 9998);`
		(4)

TAB. 10.8 – Cas de tests produits pour la méthode VERIFY_PIN

étapes, sont la cause de cette déficience sur notre exemple.

Par extension, produire une séquence de test visant à bloquer le code PIN et à le débloquer, correspondant ainsi au deuxième scénario construit avec l'animateur, est difficilement atteignable en temps raisonnable, de par la combinatoire des comportements possibles et les nombreux minima locaux qui parsèment le chemin menant à l'objectif. Les temps de calcul résultants étant trop longs, deux solutions s'offrent à nous : soit adopter un autre algorithme pour la génération de tests, soit proposer une solution alternative. L'évaluation d'autres algorithmes de recherche, telles que des mécanismes de chaînage arrière, a déjà été abordée durant les travaux de thèse de Mlle Séverine Colin [Col05]. Pour le problème qui nous intéresse, la construction des séquences de test par chaînage arrière se heurte au même problème des minima locaux lors de la recherche. Nous proposons donc une solution alternative, mettant en œuvre une technique semi-automatique, qui rejoint l'animation symbolique et ouvre des perspectives intéressantes sur le guidage de la génération de tests.

Calcul guidé

Pour pallier aux problèmes d'atteignabilité des cibles de tests, nous proposons un mécanisme couplant l'animation et la génération automatique de tests. Ce mécanisme a pour but d'offrir la possibilité à un utilisateur d'aider à la construction de la séquence de tests. Ce mécanisme consiste à :

- définir, par l'utilisateur, une séquence initiale potentiellement contrainte définissant un état contraint autre que l'état initial ;
- donner la main aux algorithmes de recherche systématiques afin d'atteindre les cibles de tests.

Ainsi, en définissant la séquence d'exécution symbolique donnée en figure 10.2, dans la colonne de gauche, où les caractères "?" symbolisent des valeurs symboliques pour les paramètres, la séquence de tests converge plus rapidement vers l'objectif fixé. L'application d'un tel mécanisme revient à définir un nouvel état initial. Par la suite, les valeurs symboliques des paramètres pourront être instanciées suivant les besoins identifiés par la cible de test. Ainsi, l'utilisation de cette séquence en guise d'aide au préambule permet d'atteindre les buts aux limites décrits dans le tableau 10.6. Par exemple, la figure

```
Demoney d = new Demoney();
d.INITIALIZE_UPDATE((byte) 3,(byte) 1);
d.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);
d.PUT_DATA((byte) 1,(byte) 0,(short) ?);
d.PUT_DATA((byte) 2,(byte) 0,(short) ?);
d.PUT_DATA((byte) 3,(byte) 3,(short) ?);
d.STORE_DATA((byte) 80,(byte) 0);
```

```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3,(byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);
var0.PUT_DATA((byte) 1,(byte) 0,(short) 32767);
var0.PUT_DATA((byte) 2,(byte) 0,(short) 1);
var0.PUT_DATA((byte) 3,(byte) 3,(short) 0);
var0.STORE_DATA((byte) 80,(byte) 0);
var0.INITIALIZE_UPDATE((byte) 2,(byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1,(byte) 0);
var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 0);
var0.INITIALIZE_TRANSACTION((byte) 2, (byte) 0,
                                    (short) 32767);
```

FIG. 10.2 – Un exemple de cas de test Java utilisant l'aide au préambule

10.2, dans la colonne de droite, donne le cas de test correspondant au comportement cpt_3 maximisé.

L'avantage est que cette approche permet de travailler aussi bien avec un préambule entièrement valué, qu'avec un préambule contenant des éléments symboliques. Cette combinaison permet d'atteindre toutes les cibles de test calculées, en réduisant la profondeur de recherche et, par conséquent le temps de calcul (exponentiel par rapport à la profondeur).

Bilan

Le nombre total de cibles de tests est récapitulé dans le tableau 10.9 pour les comportement normaux de chaque méthode, en fonction de chaque réécriture. Ils sont présentés sous la forme $A/B/C$ où A désigne le nombre de cibles de tests, B désigne le nombre de cibles consistantes et C désigne le nombre de cibles de tests atteignables.

Méthode	Réécriture 1	Réécriture 2	Réécriture 3	Réécriture 4
PUT_DATA	6/6/6			
STORE_DATA	1/1/1			
INITIALIZE_UPDATE	5/5/5	13/13/13	13/13/13	14/13/13
EXTERNAL_AUTHENTICATE	2/2/2	4/4/4	4/4/4	5/4/4
INITIALIZE_TRANSACTION	6/6/6			
COMPLETE_TRANSACTION	2/2/2			
VERIFY_PIN	4/4/4			
PIN_CHANGE_UNBLOCK	4/4/4			

TAB. 10.9 – Nombre de cibles de tests pour les comportements normaux en fonction des réécritures

Pour terminer, nous décrivons l'exécution des cas de tests sur une implantation de l'exemple. Nous illustrons ainsi quelques erreurs qu'il est possible de détecter de par l'exécution des tests produits.

10.3.3 Exécution des tests

Pour illustrer le passage des cas de tests, nous considérons une implantation erronée des méthodes VERIFY_PIN (pour laquelle les cas de tests ont été calculés automatiquement) et PIN_CHANGE_UNBLOCK (en utilisant le cas de test précédent). Les tests sont exécutées sur un implantation annotée par les modèles JML. La vérification des annotation fournira l'oracle lors de l'exécution du test.

Considérons l'implantation erronée de VERIFY_PIN ci-après. Pour simplifier, les vérifications des valeurs des paramètres et de l'état du système ne sont pas affichées. La portion de code représentée ne s'intéresse qu'à la zone où se trouve les erreurs.

```
public void VERIFY_PIN(byte p1, byte p2, short data) {

    // vérification des paramètres p1, p2, data
    ...
```

```
    // vérification des attributs personnalized, accessLevel
    ...

    if (pin.tries >= 0) {
       if (pin.pin == data) {
           pin.validated = true;
       }
       else {
           pin.tries--;
       }
    }
    else {
       throw new WrongParameterException();
    }
}
```

Nous exerçons cette méthode avec les 4 cas de tests précédemment calculés et donnés en figure 10.8. Les résultats de l'exécution des tests sur cette méthode Java erronée sont donnés par le tableau suivant :

Cas de test	Verdict du test	Exception
(1)	Echec	JMLNormalPostconditionError
(2)	Succès	
(3)	Succès	
(4)	Succès	

L'erreur détectée par le cas de test (1) illustre une incohérence du code par rapport au modèle puisque le programme n'effectue pas la remise du nombre d'essais du code PIN à sa valeur maximale. La postcondition spécifiée n'est ainsi pas vérifiée, ce qui est indiqué par l'exception suivante :

```
org.jmlspecs.jmlrac.runtime.JMLInternalNormalPostconditionError: by method
Demoney.VERIFY_PIN regarding specifications at
File "Demoney.java", line 411, character 40 when
        'this' is Demoney@19f953d
        at Demoney.VERIFY_PIN(Demoney.java:11163)
        at test_verify_pin.main(test_verify_pin.java:18)
```

Cette erreur se corrige en ajoutant l'instruction pertinente, qui réinitialise le nombre d'essais du code PIN si le code PIN est vérifié :

```
        if (pin.pin == data) {
            pin.validated = true;
            pin.tries = pin.triesLimit;   // correction du bug
        }
```

Considérons à présent l'implantation de la méthode PIN_CHANGE_UNBLOCK ci-dessous.

```
public void PIN_CHANGE_UNBLOCK(byte P1, byte P2, short data) {

    // vérification des paramètres P1, data

    // vérification des attributs personnalized, accessLevel

    if (pin.tries == 0 && P2 == PCU_P2) {
        pin.tries = pin.triesLimit;
    }
    else {
        if (P2 >= OwnerPIN.PIN_MIN_TRY && P2 <= OwnerPIN.PIN_MAX_TRY) {
           pin.triesLimit = P2;
           pin.tries = P2;
           pin.pin = data;
        }
    }
}
```

L'exécution du scénario spécifiquement défini à l'animateur permet d'isoler une erreur, non pas sur la méthode PIN_CHANGE_UNBLOCK, bien qu'elle soit la cible du test, mais sur la méthode VERIFY_PIN précédemment corrigée. En effet, suite aux trois essais erronés réalisés dans la séquence de test, le programme autorise la vérification et la validation d'un code PIN correct. Cette erreur est due à la condition `(pin.tries >= 0)` qui précède l'étape de vérification du code, et qui autorise le code PIN à être vérifié même lorsque les essais sont épuisés. Pour corriger cette erreur, il est nécessaire que le code PIN ne soit vérifié que si `pin.tries > 0`.

Nous remarquons également, à propos de l'implantation de la méthode PIN_CHANGE_UNBLOCK, qu'un cas de test visant à saisir une valeur erronée pour le paramètre `P2` ou une valeur erronée pour `pin.tries` produit également une erreur : le programme ne déclenchera pas d'exception de type `WrongParameterException` comme précisé dans le modèle.

10.4 Synthèse

Ce chapitre a présenté l'application des techniques décrites dans ce mémoire sur un exemple réaliste issu d'une spécification industrielle d'une applet JavaCard. Celle-ci a été adaptée pour se conformer aux spécifités d'entrée de JML-Testing-Tools.

La procédure de validation de l'exemple avec l'animateur JML-Testing-Tools a permis d'établir un modèle utilisable pour la génération de tests. De plus, la création de séquences d'animation permet la définition de cas de tests qui ne sont pas nécessairement ciblés par l'approche de génération automatique.

Néanmoins le calcul des séquences de tests permettant d'atteindre les objectifs fixés n'étant pas pleinement satisfaisant (en terme de temps de calcul et de couverture des cibles atteintes), nous avons proposé la possibilité d'alléger la tâche, et ainsi le temps d'exécution, du calcul de préambule. Pour ce faire, nous considérons comme aide au préambule des séquences d'exécutions calculées par l'utilisateur en utilisant l'animateur symbolique.

La combinaison de l'animateur et du générateur de test se révèle intéressante et permet de générer des séquences de tests plus complexes sans demander un effort trop important

de la part de l'utilisateur.

L'application de la méthode sur l'étude de cas s'est révélée pertinente et clôture la description des travaux menés dans cette thèse. Nous présentons nos conclusions sur ces contributions au chapitre suivant, puis nous terminerons par la mise en évidence des perspectives qui se dégagent de ces travaux.

Quatrième partie

Conclusion et perspectives

Conclusion

A l'heure actuelle, les langages de modélisation basés sur des annotations prennent une importance croissante. Un des indicateurs de cet intérêt est illustré par les efforts développés par Microsoft, le leader mondial de l'édition de logiciels, pour développer son propre langage d'annotations, SPEC# [MBS04], destiné à annoter les programmes C# [Tro01], dont Microsoft est propriétaire. L'intégration de SPEC# dans l'environnement de développement Visual Studio témoigne de la volonté du géant de Redmond de voir se démocratiser l'emploi de ce type de langages. Ces langages de modélisation, qui embarquent le modèle à l'intérieur du programme, permettent à ces deux entités, jadis séparées, de désormais partager le même niveau d'abstraction. L'intérêt est évident pour des activités de confrontation entre le programme et la spécification, comme pour le calcul et l'exécution de tests, puisque les séquences de test produites à partir du modèle ne nécessitent pas ou peu d'adaptation pour être exécutées sur le programme. De plus, la vérification des assertions offre un oracle gratuit lors du passage des tests sur l'implantation.

Nous avons choisi d'appliquer ces méthodes à un langage de modélisation basé sur des annotations embarquées dans un programme Java, et écrites en Java Modeling Langage (JML). Les travaux présentés dans ce mémoire se sont intéressés à l'utilisation de JML pour la validation par évaluation symbolique à contraintes. La phase de validation intervient à deux niveaux : tout d'abord, la validation d'un modèle par animation, puis la validation d'un programme en générant des tests à partir du modèle.

Nous nous sommes dans un premier temps intéressés à la validation de modèles formels objets écrits en JML. Pour ce faire, nous sommes partis d'un environnement d'animation et de résolution de contraintes logico-ensembliste, initialement conçu pour la prise en compte de spécifications B, et basé sur un format intermédiaire nommé BZP. Nous avons proposé une représentation des états mémoire Java dans ce format ensembliste, permettant ainsi leur expression sous la forme de systèmes de contraintes. Nous avons ensuite défini l'expression de spécifications JML dans le format BZP, qui nous a permis de pratiquer l'animation de la spécification.

Une fois la représentation symbolique et l'animation des spécifications Java/JML réalisées, nous nous sommes intéressés à la génération de tests aux limites pour les programmes Java à partir de leur spécification JML. Nous avons ainsi défini les objectifs de tests comme étant l'activation des comportements extraits des spécifications JML des méthodes et nous avons défini différents critères de couverture pour la spécification. Nous avons à cette occasion défini la notion de "valeur aux limites" pour des objets.

Nous avons ensuite proposé une méthode pour combler le fossé entre un modèle "validé", par animation, et un modèle "prêt pour la génération de tests". A cet égard, nous nous sommes intéressés à la vérification de la cohérence et de la consistance d'un modèle JML. Nous nous sommes également focalisés sur un mécanisme de détection automatique d'imprécisions dans le modèle JML qui peuvent nuire à la qualité des tests générés.

L'intégralité de cette approche, de l'animation symbolique à la génération de tests a été implantée dans un prototype nommé JML-Testing-Tools.

Pour finir, nous avons expérimenté notre approche sur une étude de cas issue du monde de la JavaCard, décrivant un porte-monnaie électronique. Cette expérimentation a révélé l'intérêt de l'animateur JML-Testing-Tools dans le processus d'écriture du modèle JML. En effet, son utilisation a aidé à la détection de différentes erreurs de conception, même des plus subtiles liées au codage des informations.

La génération de tests sur le modèle validé a pu être mise en œuvre. Si les objectifs de tests calculés se sont montrés pertinents, le calcul automatique des séquences de tests a montré ses limites, à cause de la physionomie de l'exemple traité. Pour pallier ces limites, nous avons proposé un mécanisme générique d'aide au calcul des séquences de test en s'appuyant sur l'utilisation de l'animateur JML-Testing-Tools pour calculer de manière semi-automatique une partie du préambule constituant le cas de test. Cette aide permet ainsi à l'utilisateur, et à moindre effort, de pouvoir produire les séquences de tests atteignant les objectifs précédemment fixés.

L'approche que nous avons présentée dans ce mémoire est satisfaisante du point de vue des résultats produits, mais elle est assez contraignante du point de vue des conditions pour sa mise en œuvre. En effet, elle nécessite que les annotations du modèle, et en particulier les postconditions de méthodes, soient suffisamment complètes pour permettre l'animation de la spécification. Il tombe sous le sens que l'application de ces procédés ne peut s'appliquer à n'importe quel programme Java.

En effet, la plupart des utilisateurs de JML se concentrent sur la description des invariants de classe, des contraintes historiques et des préconditions de méthodes, avec les objectifs respectifs de vérifier les propriétés lors de l'exécution et de filtrer les appels de méthodes illicites. Dans cette optique, les modèles produits se destinent à la génération de données de tests pour des approches de test unitaire des méthodes annotées. Les postconditions sont considérées comme oracle ; elles ne sont donc pas suffisamment complètes pour permettre l'animation. Dans ce cadre, la représentation symbolique que nous avons définie peut s'appliquer et permettre le calcul de données de tests aux limites. Sous l'hypothèse de l'accès aux différents champs des objets pour permettre de fixer leurs valeurs, nous sommes en mesure de produire des cas de tests, selon le principe de Korat [BKM02].

En outre, si un effort de modélisation supplémentaire est porté sur les postconditions, de nouveaux types de tests peuvent être réalisés, couvrant les comportements des méthodes à travers des séquences de test plus élaborées. C'est ce qu'illustre l'approche présentée dans ce mémoire.

Dans le cadre de systèmes critiques, et en particulier dans le domaine de la carte à puce et des programmes JavaCard, il n'est pas irréaliste de demander aux ingénieurs validation l'effort de concevoir un modèle complet, qui soit quasiment la réplique du programme an-

noté. Les travaux présentés dans cette thèse peuvent ainsi trouver leur application dans ce domaine. Les solutions apportées dans cette thèse vont en ce sens, en aidant également à la conception de modèles fins, par analyse symbolique de ces derniers.

Les structures de données actuellement supportées sont suffisantes pour prendre en compte des modèles réalistes, mais elles nécessitent des adaptations pour utiliser notre approche. Pour pallier ce problème, il est nécessaire de procéder à des évolutions, à différents niveaux. Ces évolutions, ainsi que les perspectives qui ressortent de ces travaux, sont présentées au chapitre suivant.

Perspectives

Les perspectives qui se dégagent de ces travaux se classent en trois parties. La première concerne les perspectives d'évolution des traitements/développements réalisés dans cette thèse. La seconde partie concerne les perspectives à court terme, liés à la génération de tests. La dernière partie offre une réflexion sur l'avenir des travaux présentés dans ce mémoire.

1 Extension des travaux réalisés

Les perspectives d'évolution sont relatives aux différents concepts et développements proposés dans cette thèse. La première évolution qui s'impose naturellement est l'extension de la couverture des structures de données manipulées, aussi bien au niveau de Java, qu'au niveau de JML. L'évolution suivante est la prise en charge des mécanismes natifs de la JavaCard, qui faciliterait l'application des méthodes proposées au milieu industriel.

1.1 Extension des structures Java supportées

Les structures de données supportées par l'approche ont limité les expérimentations, mais elles ont tout de même permis de l'appliquer, moyennant quelques adaptations mineures, à un exemple de taille réaliste. Il serait néanmoins pertinent de diminuer les efforts d'adaptation en intégrant le support natif de différents types de données supplémentaires. Celle-ci requiert la capacité à gérer des valeurs symboliques pour ces structures, nécessitant, de ce fait, des solveurs dédiés.

La plus pertinente des évolutions serait l'intégration des tableaux qui permettrait d'élargir la capacité de traitement des structures Java. Pour ce faire, des solutions à base de fonctions totales ont été étudiées au début de ces travaux de thèse. Malheureusement, le solveur de contraintes CLPS-BZ n'est pas optimisé pour le traitement des fonctions et d'ensembles d'ensembles ; le développement de ces fonctionnalités a donc été suspendu. L'utilisation d'un solveur dédié aux tableaux permettrait de continuer les travaux initiés.

Une autre limitation liée aux types de données supportés vient du fait que notre approche travaille avec deux langages sous-jacents différents. Nous sommes confrontés à des disparités entre les échelles de valeurs des différents types d'entiers. Dans notre approche, les entiers Java, codés sur 32 ou 64 bits ne peuvent être traduits que par des entiers codés sur 24 bits. Cette restriction dépend de la machine virtuelle de SICStus Prolog sur laquelle s'exécute le moteur d'interprétation des spécifications. Une solution à ce problème serait alors de migrer les solveurs de contraintes dans un environnement Java natif, comme par

exemple en utilisant le Java Contraint Kit [AKSS01] (permettant la définition de règles de gestion de contraintes en Java), ou Kaolog [Kao05] (un solveur de contraintes en Java), ou encore Cream [Cre05] ou JCL [JCL05] (des bibliothèques destinées à la programmation par contraintes en Java), ou Choco [Cho05]. Cette alternative permettrait de poser des contraintes sur des types de données basiques compatibles entre le langage source, Java, et le langage de contraintes utilisé pour l'interprétation des spécifications. Cette solution est à nuancer, car la compatibilité de ces types basiques ne dispense pas de concevoir une modélisation des états mémoire Java permettant la manipulation des objets. Celle-ci devrait alors être réalisée dans le solveur choisi, soit en utilisant les structures de données offertes par le langage, soit en développant un mécanisme adhoc basé sur des contraintes logico-ensemblistes comme c'est le cas des travaux présentés dans ce mémoire.

1.2 Extension de la couverture du JML

L'extension de la couverture de la notation JML est également envisageable. Pour l'instant, un sous-ensemble suffisamment expressif est supporté, mais on peut imaginer le support de mots-clés supplémentaires. On peut par exemple s'intéresser aux mot-clés faisant des références aux structures de données internes de Java, notamment `\reach(o)` qui permet de collecter l'ensemble des objets accessibles à travers les attributs d'un objet `o` donné. La représentation explicite des états Java que nous réalisons permet aisément ce genre d'évolution.

Une autre extension aisément envisageable est de surcharger certaines classes d'objets JML pour en présenter le support natif. Ainsi, les classes *JMLObjectSet* et *JMLObjectValue*, représentant respectivement des ensembles d'objets et de valeurs, pourraient être surchargées pour être directement interprétées par la partie ensembliste du solveur CLPS-BZ.

1.3 Adaptation au mécanisme de la JavaCard

L'expérimentation menée sur une application issue du monde JavaCard a nécessité quelques adaptations : des adaptations pour la gestion des structures de données (tableaux), mais également des adaptations liées au mécanisme de la JavaCard en lui-même, pour faciliter le calcul des séquences de test.

Une évolution possible serait le développement d'un mécanisme interne, lors de la génération des séquences de test, pour la création systématique de paramètres APDU adéquats, encapsulant les paramètres effectifs des méthodes JavaCard. Une autre alternative est d'adapter le mécanisme de réification qui permet de faire cette transcription et rend possible le passage de tests sur une implantation.

2 Évolutions sur la méthode de test

Nous présentons ici les perspectives à court terme qui découlent des applications directes des travaux réalisés. Ces perspectives sont relatives à l'amélioration de la méthode proposée pour détecter des erreurs dans un programme. Pour ce faire, deux possibilités s'offrent à nous : soit modifier la méthode de test, soit envisager une autre approche.

2.1 Test guidé par des propriétés

La définition de politiques de sécurité et les exigences définies sur les applications à valider, tendent à faire évoluer le test vers une approche semi-automatique, guidée par différents critères.

Le mécanisme de génération de tests que nous avons proposé peut être adapté pour générer des séquences de test à partir de propriétés temporelles. On peut citer les travaux de M. Huisman et K. Trentleman dans [TH02], qui traitent de l'expression de propriétés temporelles pour programmes JavaCard sous la forme d'annotations JML supplémentaires. Pour ce faire, ils proposent un langage de plus haut niveau, nommé JTPL, qui est actuellement mis en œuvre dans le cadre de la thèse au LIFC de M. Julien Groslambert, encadrée par M. Jacques Julliand et M. Alain Giorgetti. Ces travaux ont mis en évidence l'utilisation de ce langage dans le cadre de l'expression et la vérification de propriétés de sûreté et de vivacité [BGH+04]. Pour ce faire, le langage exprime des propriétés qui se doivent d'être vérifiées suite au déclanchement de différents événements (appels de méthodes, terminaison normales ou exceptionnelles). Chaque propriété temporelle produit des annotations JML supplémentaires sous la forme d'invariants ou de contraintes historiques, qui servent à enrichir le modèle JML sur lequel elles sont exprimées. L'utilisation de ces propriétés temporelles permet de définir et d'exprimer sur le programme les *politiques de sécurité*, dont les concepts ont émergé récemment. La perspective d'évolution proposée ici est d'employer les propriétés temporelles pour guider la génération de test à partir du modèle, de manière à produire des cas de test qui couvrent une propriété de sûreté. De plus, les annotations relatives à la propriété temporelle ajoutées au modèle JML renforcent l'oracle lors du passage des tests. L'outil JAG [GG06] a ainsi été développé au LIFC pour permettre la génération d'annotations JML à partir de propriétés temporelles exprimées en JTPL. Cet outil offre la traçabilité des annotations, qui permet de retrouver la propriété temporelle d'où sont issues les annotations qui n'ont pas été vérifées que ce soit par preuve, ou lors d'une exécution.

La seconde évolution nous a été inspirée par les expérimentations menées durant cette thèse. A partir de l'animation de la spécification, nous avons constaté que la construction automatique de séquences de tests n'était pas optimale à cause de certains enchaînements obligatoires de méthodes qui ralentissent le calcul. Notre solution consiste à demander à l'utilisateur de construire tout ou partie des cas de tests de manière semi-automatique. Cette approche requiert de celui-ci qu'il ait une bonne connaissance de la spécification. Ceci rejoint les pré-requis à l'utilisation efficace d'un outil de génération de test combinatoire, comme par exemple Tobias [LdBMB04]. Une perspective d'évolution en ce sens serait l'application d'une méthode de génération de test combinatoire basée sur un schéma de test qui engendrerait par son dépliage la construction de séquences d'exécutions en utilisant l'animation symbolique de la spécification. Cette approche présente l'avantage de filtrer les séquences de tests à la construction pour ne garder que celles satisfaisant les spécifications JML. Là encore, l'utilisation de schémas de tests permet à un utilisateur de concevoir des jeux de tests pilotés plus finement qui satisfont la politique de sécurité ou les exigences imposées.

2.2 Test de non-conformité

La richesse du modèle JML est la clé de l'utilisation de notre approche. Néanmoins, cette nécessité restreint son application à des cadres bien précis de logiciels embarqués qui nécessitent un haut niveau de confiance. Nous souhaiterions donc permettre d'utiliser tout ou partie des travaux réalisés dans le contexte plus général des programmes Java. Une des perspectives à ces travaux est offerte par l'intermédiaire du projet DANOCOPS.

Le projet RNTL DANOCOPS[12] [Dan05] est la suite du projet RNTL InKA [GCR96, GBR98, Got00]. Ce dernier visait à la production de cas de tests structurels de code C, basé sur la résolution de contraintes. Cette suite s'intéresse à détecter différents types de non-conformités entre le programme et sa spécification, en employant des mécanismes de résolution de contraintes. Les différents types de non-conformités à détecter ont été identifiés comme étant les suivants :

- *Non-conformité fournisseur* : celle-ci consiste à vérifier qu'une classe respecte toujours son contrat de fournisseur ; ce type de non-conformité est détecté si dans des conditions licites d'utilisation, une méthode ne fournit pas le service attendu (violation des postconditions).
- *Non-conformité client* : celle-ci consiste à vérifier que chaque classe utilise les autres en respectant ses obligations de client ; ce type de non-conformité est détecté si une méthode ne remplit pas les conditions d'utilisation lors de l'invocation interne d'une autre méthode (violation des préconditions de la méthode appelée).
- *Non-conformité d'intégrité* : celle-ci consiste à vérifier que les objets des classes conservent leur intégrité ; ce type de non-conformité est détecté si une méthode ne respecte pas son invariant.

La détection de ces non-conformités est réalisée par confrontation du système de contraintes issu du code Java et le système de contraintes issu de la spécification JML en cherchant à réfuter une propriété issue du modèle. A cette occasion, une collaboration est mise en place entre les solveurs InKA, en charge de gérer les contraintes issues de l'exécution symbolique du programme Java, et le solveur CLPS-BZ, en charge de gérer les contraintes spécifiées dans le modèle JML.

3 JML-Testing-Tools, vers un outil industriel ?

Les développements et les expérimentations effectués dans cette thèse se sont destinés à une vocation purement académique. En s'appuyant sur l'évolution de la technologie BZ-Testing-Tools, il est intéressant de se demander si les travaux présentés ici peuvent se voir appliqués au milieu industriel, avec toutes les implications sous-entendues : passage à l'échelle, adoption de la méthode dans le processus de développement, intégration dans des environnements de développements, etc.

À l'heure actuelle, la technologie développée est capable de gérer des applications de tailles réalistes mais le passage à l'échelle n'est pas pour autant assuré. Outre les soucis d'ordre techniques, tels que les optimisations des traitements des solveurs et l'élargissement des structures supportées, le passage à l'échelle pose le problème de l'explosion

[12] Détection Automatique de NOn-COnformité entre un Programme et sa Spécification

combinatoire. Si celle-ci est retardée pour les états du système par l'introduction de valeurs symboliques, le nombre de transitions potentiellement activable reste un challenge, et un obstacle, à l'efficacité du calcul de préambule (de complexité exponentielle). Malheureusement, ce facteur entre en jeu dans le calcul automatique des séquences de tests et des solutions adéquat doivent être envisagées.

Pour résoudre ce problème, une solution pertinente serait de "diviser pour mieux régner" et de considérer la décomposition du système de grande taille en différents composants de taille raisonnable. Ce type d'approche est assez adapté à Java/JML, puisque les classes d'objets décrivent des composants qui peuvent la plupart du temps être validés sans avoir à considérer l'intégralité du système.

Comme nous l'avons évoqué au cours de la conclusion de ces travaux, l'application des résultats de cette thèse au monde industriel nécessite de concevoir des modèles JML suffisamment complets. L'écriture de tels modèles a pour conséquence de quasiment dupliquer le code du programme, ce qui nécessite un effort considérable de la part du spécifieur et restreint son emploi aux systèmes critiques. L'emploi de JML dans le domaine de la JavaCard est un fait intéressant, car il nous renseigne sur l'intérêt et l'attractivité que représentent les langages d'annotations, puisqu'ils facilitent l'adoption des méthodes formelles par le milieu industriel. Une application des techniques présentées dans ce mémoire à l'industrie de la carte à puce est donc potentiellement réalisable.

Annexes

Annexe A

Tests produits pour l'exemple fil rouge

Nous donnons ici les cibles de test, et leurs cas de test associés, issus de l'exemple fil rouge du porte-monnaie simplifié.

A.1 Classe `Purse`

Constructeur `Purse(short)`

Cibles de test

Partie avant	P_{spe}	
`a >= 0`	`a == 0`	$(A.1)$
	`a == 32767`	$(A.2)$

Cas de test

$(A.1)$ `Purse var0 = new Purse((short) 0);`

$(A.2)$ `Purse var0 = new Purse((short) 32767);`

Méthode `credit(short)`

Cibles de test

Partie avant	P_{spe}	
`a > 0 && (balance + a) <= 32767 && \typeof(this) == \type(Purse)`	`a == 1 && balance == 0 && this.hist == null`	$(B.1)$
	`a == 1 && balance == 0 && this.hist != null && \typeof(this.hist) == \type(History)`	$(B.2)$
	`a == 32766 && balance == 1 && this.hist == null`	$(B.3)$
	`a == 32766 && balance == 1 && this.hist != null && \typeof(this.hist) == \type(History)`	$(B.4)$

Cas de test

$(B.1)$	`Purse var0 = new Purse((short) 0);` `var0.credit((short) 1);`

$(B.2)$	`Purse var0 = new Purse((short) 1);`
	`var0.debit((short) 1);`
	`var0.credit((short) 1);`

$(B.3)$	`Purse var0 = new Purse((short) 1);` `var0.credit((short) 32766);`

$(B.4)$	`Purse var0 = new Purse((short) 2);`
	`var0.debit((short) 1);`
	`var0.credit((short) 32766);`

Méthode `debit(short)`

Cibles de test

Partie avant	P_{spe}	
`a > 0 && a <= balance`	`a == 1 && balance == 2 &&` `this.hist == null`	$(C.1)$
	`a == 1 && balance == 2 &&` `this.hist != null && \typeof(this.hist) == \type(History)`	$(C.2)$
	`a == 32767 && balance == 32767 &&` `this.hist == null`	$(C.2)$
	`a == 32767 && balance == 32767 &&` `this.hist != null && \typeof(this.hist) == \type(History)`	$(C.4)$
Réécriture 1		
`a <= 0 || a > balance`	`a == -32768 && balance == 0 && hist == null`	$(C.5)$
	`a == 32767 && balance == 32766 && hist == null`	$(C.6)$
	`a == -32768 && balance == 0 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.7)$
	`a == 32767 && balance == 32766 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.8)$
Réécriture 2		
`a <= 0`	`a == -32768 && hist == null`	$(C.9)$
	`a == 0 && hist == null`	$(C.10)$
	`a == -32768 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.11)$
	`a == 0 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.12)$
`a > balance`	`a == 1 && balance == 0 && hist == null`	$(C.13)$
	`a == 32767 && balance == 32766 && hist == null`	$(C.14)$
	`a == 1 && balance == 0 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.15)$
	`a == 32767 && balance == 32766 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.16)$
Réécriture 3		
`a <= 0 && a <= balance`	`a == -32768 && hist == null`	$(C.17)$
	`a == 0 && hist == null`	$(C.18)$
	`a == -32768 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.19)$
	`a == 0 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.20)$
`a > 0 && a > balance`	`a == 1 && balance == 0 && hist == null`	$(C.21)$
	`a == 32767 && balance == 32766 && hist == null`	$(C.22)$
	`a == 1 && balance == 0 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.23)$
	`a == 32767 && balance == 32766 && hist != null &&` `\typeof(this.hist) == \type(History)`	$(C.24)$
Réécriture 4 : inconsistante		

Cas de test

$(C.1)$	`Purse var0 = new Purse((short) 2);` `var0.debit((short) 1);`

$(C.2)$	`Purse var0 = new Purse((short) 0);`
	`var0.credit((short) 2);`
	`var0.debit((short) 1);`

$(C.3)$	`Purse var0 = new Purse((short) 32767);` `var0.debit((short) 32767);`

$(C.4)$	`Purse var0 = new Purse((short) 0);`
	`var0.credit((short) 32767);`
	`var0.credit((short) 32766);`

Récriture 1 :

$(C.5)$	`Purse var0 = new Purse((short) 0);` `var0.debit((short) -32768);`
$(C.6)$	`Purse var0 = new Purse((short) 32766);` `var0.debit((short) 32767);`

$(C.7)$	`Purse var0 = new Purse((short) 1);`
	`var0.debit((short) 1);`
	`var0.debit((short) -32768);`

$(C.8)$	`Purse var0 = new Purse((short) 0);`
	`var0.credit((short) 32766);`
	`var0.debit((short) 32767);`

Réécritures 2 et 3 :

$(C.9)$ et $(C.17)$	`Purse var0 = new Purse((short) 0);` `var0.debit((short) -32768);`
$(C.10)$ et $(C.18)$	`Purse var0 = new Purse((short) 0);` `var0.debit((short) 0);`

$(C.11)$ et $(C.19)$	`Purse var0 = new Purse((short) 1);`
	`var0.debit((short) 1);`
	`var0.debit((short) -32768);`

$(C.12)$ et $(C.20)$	`Purse var0 = new Purse((short) 0);`
	`var0.credit((short) 32766);`
	`var0.debit((short) 32767);`

$(C.13)$ et $(C.21)$	`Purse var0 = new Purse((short) 0);` `var0.debit((short) 1);`
$(C.14)$ et $(C.22)$	`Purse var0 = new Purse((short) 32766);` `var0.debit((short) 32767);`

$(C.15)$ et $(C.23)$	`Purse var0 = new Purse((short) 1);`
	`var0.debit((short) 1);`
	`var0.debit((short) 1);`

$(C.16)$ et $(C.24)$	`Purse var0 = new Purse((short) 0);`
	`var0.credit((short) 32766);`
	`var0.debit((short) 32767);`

Méthode `cancel()`

Cibles de test

Partie avant	P_{spe}	
`hist != null`	`this.hist != null && \typeof(this.hist) == type(History) && this.hist.balance == 0 && this.hist.previous == null`	$(D.1)$
	`this.hist != null && \typeof(this.hist) == type(History) && this.hist.balance == 0 && this.hist.previous != null && \typeof(this.hist.previous) == \type(History)`	$(D.2)$
	`this.hist != null && \typeof(this.hist) == type(History) && this.hist.balance == 32767 && this.hist.previous == null`	$(D.3)$
	`this.hist != null && \typeof(this.hist) == type(History) && this.hist.balance == 32767 && this.hist.previous != null && \typeof(this.hist.previous) == \type(History)`	$(D.4)$

Cas de test

$(D.1)$
```
Purse var0 = new Purse((short) 0);
var0.credit((short) 1));
var0.cancel();
```

$(D.2)$
```
Purse var0 = new Purse((short) 2);
var0.debit((short) 2));
var0.credit((short) 1));
var0.cancel();
```

$(D.3)$
```
Purse var0 = new Purse((short) 0);
var0.credit((short) 1));
var0.cancel();
```

$(D.4)$
```
Purse var0 = new Purse((short) 0);
var0.credit((short) 32767));
var0.debit((short) 1));
var0.cancel();
```

A.2 Classe `LimitedPurse`

Constructeur `LimitedPurse(short)`

Cibles de test

Partie avant	P_{spe}	
`a >= 0 && a <= 10000`	`a == 0`	$(E.1)$
	`a == 10000`	$(E.2)$

Cas de test

$(E.1)$
```
LimitedPurse var0 = new LimitedPurse((short) 0);
```

$(E.2)$
```
LimitedPurse var0 = new LimitedPurse((short) 10000);
```

Méthode `credit(short)`

Cibles de test

Partie avant	P_{spe}	
`a > 0 && (balance + a) <= max &&` `\typeof(this) == \type(LimitedPurse)`	`a == 1 && balance == 0 &&` `max == 1 && this.hist == null`	$(F.1)$
	`a == 1 && balance == 0 && this.hist != null &&` `max == 1 && \typeof(this.hist) == \type(History)`	$(F.2)$
	`a == 32766 && balance == 1 &&` `max == 32767 && this.hist == null`	$(F.3)$
	`a == 32766 && balance == 1 && this.hist != null &&` `max == 32767 && \typeof(this.hist) == \type(History)`	$(F.4)$

Cas de test

$(F.1)$
```
LimitedPurse var0 = new LimitedPurse((short) 0);
var0.setMax((short) 1);
var0.credit((short) 1);
```

$(F.2)$
```
LimitedPurse var0 = new LimitedPurse((short) 1);
var0.debit((short) 1);
var0.setMax((short) 1);
var0.credit((short) 1);
```

$(F.3)$
```
LimitedPurse var0 = new LimitedPurse((short) 1);
var0.setMax((short) 32767);
var0.credit((short) 32766);
```

$(F.4)$
```
LimitedPurse var0 = new LimitedPurse((short) 2);
var0.debit((short) 1);
var0.setMax((short) 32767);
var0.credit((short) 32766);
```

Méthode `setMax(short)`

Cibles de test

Partie avant	P_{spe}	
`m >= 0 && m >= balance`	`m == 0 && balance == 0`	$(G.1)$
	`m == 32767 && balance == 32767`	$(G.2)$

Cas de test

$(G.1)$
```
LimitedPurse var0 = new Purse((short) 0);
var0.setMax((short) 0);
```

$(G.2)$
```
LimitedPurse var0 = new Purse((short 0);
var0.setMax((short) 32767);
var0.credit((short) 32767);
```

Autres méthodes

Les cas de test des méthodes `debit(short)` et `cancel()` pour la classe `LimitedPurse` sont similaires à la différence près que l'objet `var0` est du type `LimitedPurse`.

Annexe B

Tests produits pour l'étude de cas

Nous donnons ici les cas de test pour les méthodes PUT_DATA et INITIALIZE_TRANSACTION pour lesquelles nous avons donné les cibles de test au chapitre 10, et que nous rappelons au préalable.

B.1 Méthode PUT_DATA

Cibles de test

Partie avant	P_{spe}	
personnalized == false && accessLevel == ADMIN_LVL && P1 == PUT_MAX_BALANCE && P2 == PUT_DATA_P2 && data > 0	data == 1	(A.1)
	data == 32767	(A.2)
personnalized == false && accessLevel == ADMIN_LVL && P1 == PUT_MAX_DEBIT && P2 == PUT_DATA_P2 && data > 0	data == 1	(A.3)
	data == 32767	(A.4)
personnalized == false && accessLevel == ADMIN_LVL && P1 == PUT_PIN && data >= 0 && data <= 9999 && P2 >= OwnerPIN.PIN_MIN_TRY && P2 <= OwnerPIN.PIN_MAX_TRY	P2 == 3 && data == 0	(A.5)
	P2 == 15 && data == 9999	(A.6)

Cas de test

(A.1)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 1, (byte) 0, (short) 1);
```

(A.2)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 1, (byte) 0, (short) 32767);
```

(A.3)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 2, (byte) 0, (short) 1);
```

($A.4$)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 2, (byte) 0, (short) 32767);
```

($A.5$)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 3, (byte) 3, (short) 0);
```

($A.6$)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 3, (byte) 15, (short) 9999);
```

B.2 Méthode INTIALIZE_TRANSACTION

Cibles de test

Partie avant	P_{spe}	
`personnalized == true && authenticateSequence == PUBLIC_LVL && transactionSequence == 0 && p2 == IT_P2 && P1 == IT_P1_DEBIT && accessLevel == DEBIT_LVL && data > 0 && data <= maxDebitAmount && data <= balance`	`data == 1 && maxDebitAmount == 1 && balance == 1`	($B.1$)
	`data == 32767 && maxDebitAmount == 32767 && balance == 32767`	($B.2$)
`personnalized == true && authenticateSequence == PUBLIC_LVL && transactionSequence == 0 && p2 == IT_P2 && P1 == IT_P1_CREDIT_FROM_CASH && accessLevel == CREDIT_LVL && data > 0 && (balance + data) <= maxBalance`	`data == 1 && maxBalance == 1 && balance == 0`	($B.3$)
	`data == 32767 && maxBalance == 32767 && balance == 0`	($B.4$)
`personnalized == true && authenticateSequence == PUBLIC_LVL && transactionSequence == 0 && p2 == IT_P2 && P1 == IT_P1_CREDIT_FROM_BANK && accessLevel == CREDIT_LVL && data > 0 && (balance + data) <= maxBalance && pin.validated == true`	`data == 1 && maxBalance == 1 && balance == 0`	($B.5$)
	`data == 32767 && maxBalance == 32767 && balance == 0`	($B.6$)

Cas de test

($B.1$)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 2, (byte) 0, (short) 1);
var0.PUT_DATA((byte) 3, (byte) 3, (short) 0);
var0.STORE_DATA((byte) 80, (byte) 0);
var0.INITIALIZE_UPDATE((byte) 2, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.INITIALIZE_TRANSACTION((byte) 1, (byte) 0, (short) 1);
var0.COMPLETE_TRANSACTION((byte) 0, (byte) 0);
var0.INITIALIZE_UPDATE((byte) 1, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.INITIALIZE_TRANSACTION((byte) 0, (byte) 0, (short) 1);
```

(B.2)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 2, (byte) 0, (short) 32767);
var0.PUT_DATA((byte) 3, (byte) 3, (short) 0);
var0.STORE_DATA((byte) 80, (byte) 0);
var0.INITIALIZE_UPDATE((byte) 2, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.INITIALIZE_TRANSACTION((byte) 1, (byte) 0, (short) 32767);
var0.COMPLETE_TRANSACTION((byte) 0, (byte) 0);
var0.INITIALIZE_UPDATE((byte) 1, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.INITIALIZE_TRANSACTION((byte) 0, (byte) 0, (short) 32767);
```

(B.3)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 2, (byte) 0, (short) 1);
var0.PUT_DATA((byte) 3, (byte) 3, (short) 0);
var0.STORE_DATA((byte) 80, (byte) 0);
var0.INITIALIZE_UPDATE((byte) 2, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.INITIALIZE_TRANSACTION((byte) 1, (byte) 0, (short) 1);
```

(B.4)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 2, (byte) 0, (short) 32767);
var0.PUT_DATA((byte) 3, (byte) 3, (short) 0);
var0.STORE_DATA((byte) 80, (byte) 0);
var0.INITIALIZE_UPDATE((byte) 2, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.INITIALIZE_TRANSACTION((byte) 1, (byte) 0, (short) 32767);
```

(B.5)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 2, (byte) 0, (short) 1);
var0.PUT_DATA((byte) 3, (byte) 3, (short) 0);
var0.STORE_DATA((byte) 80, (byte) 0);
var0.INITIALIZE_UPDATE((byte) 2, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 0);
var0.INITIALIZE_TRANSACTION((byte) 2, (byte) 0, (short) 1);
```

(B.6)
```
Demoney var0 = new Demoney();
var0.INITIALIZE_UPDATE((byte) 3, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.PUT_DATA((byte) 2, (byte) 0, (short) 32767);
var0.PUT_DATA((byte) 3, (byte) 3, (short) 0);
var0.STORE_DATA((byte) 80, (byte) 0);
var0.INITIALIZE_UPDATE((byte) 2, (byte) 1);
var0.EXTERNAL_AUTHENTICATE((byte) 1, (byte) 0);
var0.VERIFY_PIN((byte) 0, (byte) 0, (short) 0);
var0.INITIALIZE_TRANSACTION((byte) 2, (byte) 0, (short) 32767);
```

Bibliographie

[ABC+02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT : A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brnö, Czech Republic, August 2002. INRIA Technical Report.

[Abr96] J-R. Abrial. *The B-BOOK : Assigning Programs to Meanings.* Cambridge University Press, 1996. ISBN 0 521 49619 5.

[AKSS01] Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. Jack : A java constraint kit. In *Proceedings International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, Kiel, September 13-15 2001. Society of Logic Programming (GLP e.V.; http ://wdp.first.gmd.de :8080/glp/), University of Kiel ; published as Tech Report No. 2017.

[ALL94] F. Ambert, B. Legeard, and E. Legros. Constraint Logic Programming on Sets and Multisets. In *Proceedings of the ILPS'94 Workshop on Constraint Languages/Systems and their Use in Problem Modeling*, pages 151–165, Ithaca, New York, November 1994.

[Amb97] F. Ambert. *Résolution de contraintes ensemblistes et approche multi-solveurs.* PhD thesis, LIFC - University of Franche-Comté, 1997.

[Apt03] K.R. Apt. *Principles of Constraint Programming.* Cambridge University Press, Cambridge, UK, 2003.

[BB04] Clark Barrett and Sergey Berezin. CVC Lite : A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BDG05] F. Bouquet, F. Dadeau, and J. Groslambert. Checking JML Specifications with B Machines. In *Proceedings of the International Conference on Formal Specification and Development in Z and B (ZB'05)*, volume 3455 of *LNCS*, pages 435–454, Guildford, United Kingdom, April 2005. Springer-Verlag.

[BDL05a] F. Bouquet, F. Dadeau, and B. Legeard. How symbolic animation can help designing an efficient formal model. In *Procs of the 7th Int. Conf. on Formal Engineering Methods (ICFEM'05)*, volume 3785 of *LNCS*, pages 96–110, Manchester, UK, November 2005. Springer-Verlag.

[BDL05b] F. Bouquet, F. Dadeau, and B. Legeard. Using constraint logic programming for the symbolic animation of formal models. In J. Marques-Silva and M. Velev, editors, *Procs of the Int. Workshop on Constraints in Formal Verification (CFV'05) – Co-located with the Int. Conf. on Automated Deduction (CADE'05)*, pages 32–46, Tallinn, Estonia, July 2005.

[BDL06a] F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In *Proceedings of the 14th International Conference on Formal Methods (FM'06)*, LNCS, Hamilton, Canada, August 2006. Springer-Verlag. To appear.

[BDL06b] F. Bouquet, F. Dadeau, and B. Legeard. JML-Testing-Tools : un animateur symbolique de spécifications JML. In Ecole Nationale Supérieure des Télécommunications, editor, *Sessions Outils, Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'06*, pages 373–376, Paris, France, March 2006.

[BDLU05a] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-Testing-Tools : a Symbolic Animator for JML Specifications using CLP. In *Proceedings of 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Tool session (TACAS'05)*, volume 3440 of *LNCS*, pages 551–556, Edinburgh, United Kingdom, April 2005. Springer-Verlag.

[BDLU05b] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic Animation of JML Specifications. In *Proceedings of the International Conference on Formal Methods (FM'05)*, volume 3582 of *LNCS*, pages 75–90, Newcastle, United Kingdom, July 2005. Springer-Verlag. To appear.

[Bei95] B. Beizer. *Black-Box Testing : Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, New York, USA, 1995.

[BFG+99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF : An intermediate representation and validation environment for timed asynchronous systems. In *World Congress on Formal Methods (1)*, pages 307–327, 1999.

[BFG+03] C. Bigot, A. Faivre, J-P. Gallois, A. Lapitre, D. Lugato, J-Y. Pierron, and N. Rapin. Automatic Test Generation with AGATHA. In *Proceedings of the ETAPS'03 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 591–596. Springer Verlag, 2003.

[BG94] Andreas Blass and Yuri Gurevich. Evolving algebras and linear time hierarchy. In *IFIP Congress (1)*, pages 383–390, 1994.

[BGH+04] F. Bellegarde, J. Groslambert, M. Huisman, O. Kouchnarenko, and J. Julliand. Verification of liveness properties with jml. Technical Report RR-5331, INRIA, 2004.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000 : Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[BGS+03] Michael Barnett, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Validating use-cases with the asml test tool. In *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*, pages 238–246. IEEE Computer Society, 2003.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat : automated testing based on java predicates. In *ISSTA'02 : Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM Press.

[BL03] F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation : the Java Card transaction mechanism case study. In *Proceedings of the International Conference on Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 778–795, Pisa, Italy, September 2003. Springer Verlag.

[BLL04] F. Bouquet, B. Legeard, and F. Lebeau. Test-Case and Test-Driver Generation for Automotive Embedded Software. In *Proceedings of the 5th International Conference on Software Testing (ICS-Test'04)*, pages 37 – 53, Dusseldorf, Germany, April 2004. Software and Systems Quality Conferences.

[BLLP04] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications : GSM 11.11 standard case-study. *The Journal of Software Practice and Experience*, 34(10) :915 – 948, 2004.

[BLP02] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B – A constraint solver for B. In *Proceedings of the ETAPS'02 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 188–204, Grenoble, France, April 2002. Springer Verlag.

[BMDB+01] P. Bontron, O. Maury, L. Du Bousquet, Y. Ledru, C. Oriat, and M.-L. Potet. Tobias : un environnement pour la création d'objectifs de tests à partir de schémas de tests. In *ICSSEA'2001 : Proceedings of the 13th International Conference on Software and Systems Engineering and their Applications*, Paris, France, Décembre 2001.

[Bou05] F. Bouquet. *Sémantique interprétative de spécifications formelles en animation symbolique et génération de tests à partir de modèles.* Habilitation à Diriger des Recherches, LIFC - University of Franche-Comté, 2005.

[BRL03] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness : A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003 : Formal Methods : International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[BS97] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[Bul89] Bull S.A. *Estelle Simulator/Debugger User Reference Manual.* Les Clayes s/s Bois, 1989.

[BZT05] The BZ-TT web site. `http://lifc.univ-fcomte.fr/~bztt`, 2005.

[Caf05] The CafeOBJ website. `http://www.ldl.jaist.ac.jp/cafeobj/`, 2005.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1) :7–34, 2001.

[CDD+03] J.-F. Couchot, F. Dadeau, D. Déharbe, A. Giorgetti, and S. Ranise. Proving and debugging set-based specifications. In *Electronic Notes in Theoretical Computer Science, proceedings of the Sixth Brazilian Workshop on Formal Methods (WMF'03)*, volume 95, pages 189–208, May 2003.

[CDPP96] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Softw.*, 13(5) :83–88, 1996.

[CDT01] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.2.* INRIA-Rocquencourt, December 2001.
http ://coq.inria.fr/doc-eng.html.

[CEP01] CEPSCO. *Common Electronic Purse Specifications version 2.3*, March 2001. `http://www.cepsco.com`.

[CGN+05] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Testing concurrent object-oriented systems with spec explorer. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005 : Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 542–547. Springer, 2005.

[CGP99] Edmund M. Clarke, Orna Grumberg, and A. Peled. *Model Checking.* MIT Press, 1999. CLA e 99 :1 1.Ex.

[Che05] S. Chemin. *Traitement de relations et fonctions en résolution de contraintes ensemblistes et application à l'évaluations de notations de spécifications formelles.* PhD thesis, LIFC - University of Franche-Comté, 2005.

[Cho78] T.S. Chow. Testing Software Design Modeled by Finite-State Machines. *The Journal of IEEE Transaction on Software Engineering*, 4(3), May 1978.

[Cho05] The Choco web site. `http://choco.sourceforge.net/`, 2005.

[CJRZ01] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *Proceedings of the International Conference on Research in Smart Cards (e-Smart'01)*, volume 2140 of *LNCS*, pages 58–70, Cannes, France, September 2001. Springer Verlag.

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG : A Symbolic Test Generation Tool. In *Proc. TACAS'02*, pages 470–475, 2002.

[CK04] David R. Cok and Joseph Kiniry. Esc/java2 : Uniting esc/java and jml. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128, Marseille, France, 2004. Springer.

[CL02a] Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[CL02b] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing : The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *LNCS*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[Cle01] Clearsy, Europarc de Pichaury 13856 Aix-en-Provence Cedex 3 - France. *Atelier B Technical Support version 3*, May 2001. `http://www.atelierb.societe.com`.

[CLL04] S. Colin, F. Lebeau, and B. Legeard. Génération de tests à partir de statecharts fondée sur le calcul de comportements. In *Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'04*, pages 153–167, Besançon, France, June 2004.

[Col05] S. Colin. *Procédures de recherche en génération de tests à partir de modèles de spécifications.* PhD thesis, LIFC - University of Franche-Comté, 2005.

[Cre05] The Cream web site. `http://bach.istc.kobe-u.ac.jp/cream/`, 2005.

[Dan05] The DANOCOPS web site. `http://lifc.univ-fcomte.fr/~danocops/`, 2005.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the International Conference on Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.

[DH99] Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[DNS03] D. Detlefs, G. Nelson, and J. Saxe. Simplify : A theorem prover for program checking, 2003.

[DR03] D. Déharbe and S. Ranise. Light-weight theorem proving for debugging and verifying pointer manipulating programs. In *4th International Workshop on First-Order Theorem Proving (FTP'2003)*, 2003. Extended abstract.

[ECL05] The ECLiPSe Constraint Logic Programming System. `http://www.icparc.ic.ac.uk/eclipse/`, 2005.

[EhS97] Jan Ellsberger, Dieter hogrefe, and Amardeo Sarma. *SDL Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.

[Fil03] J.-C. Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[FLL+02] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java, 2002.

[FORS01a] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS : integrated canonizer and solver. CAV'2001, 2001.

[FORS01b] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. Ics : Integrated canonizer and solver. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (Paris, France)*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, July 2001.

[GB98] E. Gamma and K. Beck. Test infected : Programmers love writing tests. *Java Report*, 3(7) :37–50, July 1998.

[GBR98] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'98)*, volume 2, pages 53–62, Clearwater Beach, USA, March 1998. ACM SIGSOFT.

[GCR96] A. Gotlieb, F. Calvet, and M. Rueher. Génération automatique de cas de test : une approche Programmation Logique par Contraintes. *Revue Génie Logiciel numéro 42*, pages 135–140, December 1996.

[GG06] Alain Giorgetti and Julien Groslambert. JAG : Jml annotation generation for verifying temporal properties. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *LNCS*, pages 373–376. Springer, 2006.

[GGSV01] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Testing with abstract state machines. In *Formal Methods and Tools for Computer Science (EUROCAST'01) - Extended Abstracts*, february 2001.

[GGSV02] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State machines from Abstract State Machines. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, volume 27, pages 112–122, Rome, Italy, July 2002. ACM SIGSOFT.

[GH93] John V. Guttag and James J. Horning, editors. *Larch : Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJK99] R. Groz, T. Jéron, and A. Kerbrat. Automated test generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec*, pages 135–152. Elsevier, June 1999.

[GJS00] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 2000. Second edition.

[GLL99] J-P Gallois, A. Lapitre, and P. Lé. Analyse de spécifications industrielles et génération automatique de tests. In *Proceedings of the ICSSEA'99*, volume 2, pages 8–10. CNAM-Paris, December 1999.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Got00] A. Gotlieb. *Génération automatique de cas de test structurel avec la programmation logique par contraintes*. PhD thesis, University of Nice-Sophia Antipolis, 2000.

[GS00] J. Gray and S. Schach. Constraint Animation Using an Object-Oriented Declarative Language. In *Proceedings of the 38th Annual ACM SE Conference*, Clemson, April 2000.

[Gur05] Gurevich, Y. and Grieskamp, W. and Shulte, W. and Tillmann, N. and Veanes, M. The Microsoft AsmL site. `http://research.microsoft.com/fse/asml`, 2005.

[Har87] D. Harel. Statecharts : a Visual Formalism for Complex Systems. *Journal of Science of Computer Programming*, 8 :231–274, 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[HdR04] Grégoire Hamon, Leonardo deMoura, and John Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, September 2004. IEEE Computer Society.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to hytech. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995, Proceedings*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer, 1995.

[HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3) :231–261, 1996.

[HM00] Thomas A. Henzinger and Rupak Majumdar. A classification of symbolic transition systems. *Lecture Notes in Computer Science*, 1770 :13–44, 2000.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *The Journal of ACM Transaction on Software Engineering and Methodologies*, 5(4) :293–333, October 1996.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[Hol91] G. Holzmann. *Design and validation of protocols.* Prentice-Hall Software Series, 1991.

[HOS97] M. A. Hewitt, C. O'Halloran, and C.T. Sennett. Experiences with piza, an animator for z. In *Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 37–51. Springer-Verlag, 1997.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4) :366–381, 2000.

[HST97] D. Hazel, P. Strooper, and O. Traynor. Possum : An Animator for the SUM Specification Language. In *Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, pages 42–51, 1997.

[IL005] Statemate Tool. `http://www.ilogix.com`, 2005.

[Jac02] Daniel Jackson. Alloy : a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2) :256–290, 2002.

[JCL05] The Java Constraint Library web site. `http://liawww.epfl.ch/JCL`, 2005.

[JHA+96] J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP : a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[JHGP99] Jean-Marc Jézéquel, Wai-Ming Ho, Alain Le Guennec, and François Pennaneac'h. UMLAUT : an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[Jia95] X. Jia. An approach to animating Z specifications. In *Proceedings of the 19th Annual IEEE International Computer Software and ApplicationConference (COMPSAC 1995)*, pages 108–113, August 1995.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming : A survey. *Journal of Logic Programming*, 19/20 :503–582, May/July 1994.

[JM99] T. Jeron and P. Morel. Test Generation Derived from Model-Checking. In *Computer Aidded Verification (CAV)*, volume 1633 of *LNCS*, pages 108–121. Springer Verlag, July 1999.

[JML05] The Java Modeling Language web site. `http://jmlspecs.org`, 2005.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 2^{nd} edition, 1990.

[Jér02] T. Jéron. Tgv : théorie, principes et algorithmes. *Techniques et Sciences Informatiques, numéro spécial Test de Logiciels*, 2002.

[JSS00] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa : the alloy constraint analyzer. In *ICSE '00 : Proceedings of the 22nd international conference on Software engineering*, pages 730–733, New York, NY, USA, 2000. ACM Press.

[Kao05] The Kaolog web site. `http://www.koalog.com`, 2005.

[KKTW] Ed Kazmierczak, Peter Kearney, Owen Traynor, and Li Wang. A modular extension to z for specification, reasoning and refinement.

[K.L92] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie-Mellon University, 1992.

[KM04] Sarfraz Khurshid and Darko Marinov. Testera : Specification-based testing of java programs using sat. *Automated Software Engineering*, 11(4) :403–434, 2004.

[KMJ02] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. *SIGPLAN Not.*, 37(11) :231–245, 2002.

[KMP+96] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisma, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, 1996. `http://www.cs.umd.edu/projects/omega`.

[Kne89] R. Kneuper. *Symbolic execution as a Tool for Validation of Specifications.* PhD thesis, Department of Computer Science, University of Manchester, 1989.

[KPV03] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[LB03] M. Leuschel and M. Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[LBdB+04] Y. Ledru, P. Bontron, L. du Bousquet, O. Maury, and C. Oriat. Tobias : un outil de test combinatoire pour le test de conformité. In J. Julliand, editor, *Sessions Outils, Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'04*, pages 365–368, Besançon, France, June 2004.

[LBR98] G.T. Leavens, A.L. Baker, and C. Ruby. JML : a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998.

[LBR99] G.T. Leavens, A.L. Baker, and C Ruby. JML : A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[LBR02] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML : A behavioral interface specification language for Java. Technical Report 98-06t, Iowa State University, Department of Computer Science, December 2002. See `www.jmlspecs.org`.

[LCC+03] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. Technical Report 03-04, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, March 2003.

[LdBMB04] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering tobias combinatorial test suites. In Michel Wermelinger and Tiziana Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, pages 281–294, Barcelona, Spain, 2004. Springer-Verlag.

[Leb05] F. Lebeau. *Géneration de tests à partir de statecharts fondée sur le calcul de comportements*. PhD thesis, LIFC - University of Franche-Comté, 2005.

[LEI05] The Leirios web site. `http://www.leirios.com`, 2005.

[LP01] B. Legeard and F. Peureux. Generation of functional test sequences from B formal specifications - Presentation and industrial case-study. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE'01)*, pages 377–381, San Diego, USA, November 2001. IEEE Computer Society Press.

[LPU02] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *Proceedings of the International Conference on Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 21–40, Copenhagen, Denmark, July 2002. Springer Verlag.

[LPU04] B. Legeard, F. Peureux, and M. Utting. Controlling Test Case Explosion in Test Generation from B Formal Models. *The Journal of Software Testing, Verification and Reliability*, 14(2) :81–103, 2004.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84(2) :1090–1126, 1996.

[MA00] B. Marre and A. Arnould. Test Sequence generation from Lustre descriptions : GATEL. In *Proceedings of the 15th International Conference on Automated Software Engineering (ASE'00)*, pages 229–237, Grenoble, France, 2000. IEEE Computer Society Press.

[MBS04] K.R.M. Leino M. Barnett and W. Schulte. The Spec# Programming System : An Overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69, Marseille, France, March 2004. Springer-Verlag.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking : an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.

[MM] Renaud Marlet and Cédric Mesnil. Demoney : A demonstrative electronic purse - card specification.

[MPMU03] C. Marche, C. Paulin-Mohring, and X. Urbain. The krakatoa tool for certification of java/javacard programs annotated in jml, 2003.

[MS01] Tim Miller and Paul Strooper. Animation can show only the presence of errors, never their absence. In *ASWEC '01 : Proceedings of the 13th Australian Conference on Software Engineering*, page 76, Washington, DC, USA, 2001. IEEE Computer Society.

[MS03] T. McComb and G. Smith. Animation of Object-Z Specifications Using a Z Animator. In IEEE Computer Society, editor, *International conference on Software Engineering and Formal Methods (SEFM 2003)*, 2003.

[OMG03] OMG. *MDA Guide Version 1.0.1*, June 2003.

[Ori05] Catherine Oriat. Jartege : A tool for random generation of unit tests for java classes. In *Proccedings of the Second International Workshop on Software Quality, SOQUA 2005*, volume 3712 of *Lecture Notes in Computer Science*, pages 242–256, Erfurt, Germany, September 2005. Springer-Verlag.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. S., and M. K. Srivas. PVS : Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102, pages 411–414. Springer-Verlag, 1996.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS : a prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[OXL99] A.J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las Vegas, USA, October 1999. IEEE Computer Society Press.

[Par00] B. Parreaux. *Vérification de systèmes d'événements B par model-checking PLTL*. Phd thesis, Université de Franche-comté, Décembre 2000.

[Pau94] Lawrence C. Paulson. *Isabelle : A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[PBB+04] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Proceedings of CARDIS'04*, Toulouse, France, August 2004. Kluwer Academic Publishers.

[Peu02] F. Peureux. *Génération de tests aux limites à partir de spécifications B en Programmation Logique avec Contraintes ensemblistes*. PhD thesis, LIFC - University of Franche-Comté, 2002.

[PFTV05] Ana Paiva, João C. P. Faria, Nikolai Tillmann, and Raul F. A. M. Vidal. A model-to-implementation mapping tool for automated model-based gui testing. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, volume 3785 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2005.

[PGF+00] J-Y Pierron, J-P Gallois, E. Fievet, A. Lapitre, and D. Lugato. Validation de systèmes industriels par le test symbolique sur spécification STATEMATE. In *Proceedings of the ICSSEA'00*, volume 2, pages 5–8. CNAM-Paris, December 2000.

[PK04] Amit M. Paradkar and Tim Klinger. Automated consistency and completeness checking of testing models for interactive systems. In *28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings*, pages 342–348. IEEE Computer Society, 2004.

[Py00] L. Py. *Evaluation de spécifications formelles B en Programmation Logique avec Contraintes Ensemblistes - Application à l'animation et au model-checking.* PhD thesis, LIFC - University of Franche-Comté, 2000.

[RD03] M. Robby and J. Dwyer. Bogor : an extensible and highly-modular software model checking framework, 2003.

[RdBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *2nd International Workshop on Integrated Formal Method (IFM'00)*, pages 338–357, Dagstuhl, Germany, 2000. Springer-Verlag.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, addison-wesley edition, 1999.

[RL00] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03a, Iowa State University, 2000.

[Rob99] Harry Robinson. Graph-theory techniques in model-based testing, 1999.

[RRDH04] Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking Strong Specifications Using an Extensible Software Model Checking Framework. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420. Springer, 2004.

[Rus02] Vlad Rusu. Verification using test generation techniques. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002 : Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2391 of *Lecture Notes in Computer Science*, pages 252–271. Springer, 2002.

[SEG00] Michael Schmitt, Michael Ebner, and Jens Grabowski. Test Generation with Autolink and TestComposer. In *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM'2000), Grenoble (France), June, 26 - 28, 2000*, June 2000.

[Sof99] Softeam Europe. *Objecteering CASE Tool*, 1999.

[Spi92] J.M. Spivey. *The Z notation : A Reference Manual.* Prentice-Hall, 2^{nd} edition, 1992. ISBN 0 13 978529 9.

[SR02] Hans Schlenker and Georg Ringwelski. Pooc : A platform for object-oriented constraint programming. In Barry O'Sullivan, editor, *International Workshop on Constraint Solving and Constraint Logic Programming*, volume 2627 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 2002.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[Sun00] Sun microsystems. *Java Card 2.1.1 Virtual Machine Specification*, May 2000. `http://java.sun.com/products/javacard/javacard21.html\#specification`.

[Tel99] Telelogic. Objectgeode 4-1 reference manual, 1999.

[TH02] K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In H. Kirchner and C. Ringessein, editors, *Proceedings of AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 334–348. Springer-Verlag, 2002.

[TL05] The Trusted-Logic web site. `http://www.trusted-logic.com`, 2005.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software-Concepts and Tools*, 17(3) :103–120, 1996.

[Tro01] Andrew W. Troelsen. *C# and the .NET platform.* Apress, 2001.

[Vac04] N. Vacelet. *Évaluation de notations formelles de spécifications par systèmes de contraintes.* PhD thesis, LIFC - University of Franche-Comté, 2004.

[vdB01] Michael von der Beeck. Formalization of uml-statecharts. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 406–421. Springer, 2001.

[vdBJ01] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031 :299+, 2001.

[VPK04] Willem Visser, Corina S. Pǎsǎreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *ISSTA '04 : Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2004. ACM Press.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1998.

[XMSN05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra : A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, April 2005.

[Zha97] Hantao Zhang. SATO : an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.

Résumé

Les travaux présentés dans cette thèse se situent dans le cadre des méthodes formelles, et plus particulièrement dans le domaine de la validation de modèles et de programmes orientés objet. La validation est respectivement réalisée par animation symbolique et par des techniques de génération de tests de conformité. Les résultats présentés dans ce mémoire se focalisent sur l'interprétation à contraintes de spécifications écrites en Java Modeling Language (JML), et aux applications qui en découlent.

Ces travaux se situent dans la continuité des recherches menées dans l'équipe TFC du LIFC depuis 2001. Nous nous s'intéressons tout particulièrement à la prise en compte de la notation JML et à son intégration dans l'environnement BZ-Testing-Tools. Celle-ci a nécessité des évolutions dans le mécanisme d'interprétation à contraintes, de manière à respecter la sémantique de Java/JML. Pour cela, nous avons travaillé en deux étapes.

La première étape consiste à définir une représentation logico-ensembliste du modèle de données Java, permettant ainsi la modélisation des états d'un programme Java par un système de contraintes. Dans la seconde étape, nous proposons l'expression des spécifications de méthodes décrivant les transitions entre deux états du système.

Les résultats obtenus sur l'animation du modèle permettent de valider celui-ci dans l'objectif de produire des cas de tests fonctionnels. Pour ce faire, une méthodologie de génération de cas de tests aux limites pour un modèle objet est présentée dans ce mémoire. Ces travaux ont été implantés dans un prototype, nommé JML-Testing-Tools, qui permet l'animation symbolique d'un modèle JML et la génération de cas de tests pour l'implantation Java qui lui est associée.

Mots-clés: validation de modèle, Java Modeling Language, interprétation à contraintes, génération de tests fonctionnels

Abstract

The work presented in this thesis falls into the domain of formal methods, and especially, in the validation of object-oriented models and programs, respectively achieved by performing symbolic animation and conformance testing. The results we present in this document focus on the interpretation of formal specifications written in Java Modeling Language (JML). It also presents the applications which follow from it.

This work is made in the direction of the research led by the TFC team of the LIFC since 2001. It aims at taking into account the JML notation and integrating it within the BZ-Testing-Tools framework. It has required some evolutions within the constraint interpretation mechanism, in order to preserve the Java/JML semantics. Therefore, we have worked in two steps.

The first step consists in defining a set-thoeretic representation of the Java data model, that makes it possible to represent the states of a Java program by using constraint systems. The second step is performed by expressing the method specifications that describe the transitions between two states of the system.

The results we obtained in the animation of the JML specifications makes it possible to validate them so that they can be used for the computation of functional test cases. Therefore, a methodology is presented in this document describing the automated boundary test generation from JML specifications. This work has been implemented into a prototype, called JML-Testing-Tools, which allows the symbolic animation of a JML model and the test cases generation for its corresponding Java program.

Keywords: model validation, Java Modeling Language, constraint interpretation, functional test cases generation